ソフトウェア高速化を対象としたLLMとSLMの言語処理特性

飯塚康太[1]　吉藤尚生[1]
[1]株式会社フィックスターズ
{kota.iizuka,yoshifuji}@fixstars.com

概要

大規模言語モデル (LLM) のコスト削減を主な目的として、特定ドメインに特化した小規模言語モデル (SLM) の開発が急速に進んでいる。この論文では、ソースコードの高速化性能を題材としてLLMとSLMを比較し、SLMのコードを壊すリスクが低くなる特性が発揮される事例を示した。また、SLMとLLMを組み合わせることで互いの長所を生かしたコード高速化性能を獲得できることも示した。

1　はじめに

2022年11月にChatGPTが公開されたことをきっかけに、さまざまな企業による高性能なLLMの公開と提供が現在に至るまで盛んに行われている [1]。LLMでは数百億を超えるパラメータを持つことが多いが、最近では、特定ドメインに向けて再学習した比較的パラメータ数の少ないSLM（ドメイン特化SLM、以降単にSLM）を利用することがさまざまな領域で増加している [2][3]。しかしながら、SLMの評価は、LLMの評価手法を適用したときにどれくらいの性能劣化が起こるかという側面で検証されることが多く、SLM独自の特性についてはあまり明らかにはなっていない。例えば、コード補完などの一部のタスクではSLMとLLMで性能差が無いという報告もある [4][5] が、タスク自体が複雑でないことが主な要因で性能差が出なかったと考えられている。

また、LLMでは自然言語のみならずプログラミング言語も扱うこともでき、ソースコードを改善する応用研究も盛んである [6]。このようなソースコードの改善のうち、特に高速化（実行時間の短縮）は、同等の資源のまま計算できる量を増大させることができるため、省エネルギー・低コスト化などの面で重要な改善である。実際にLLMを用いると、例えば付録Aのような高速化事例を得ることができる。

そこで本研究では、「競技プログラミングコンテストに提出されたコードを高速化するように書き換える」というタスクを与えてSLMとLLMを比較し、特にコードの実行可能性と実行時間の短縮量という2点の特性について、比較と評価を実施した。その結果、SLMは汎用LLMと異なる長所特性を発揮することを発見した。

この論文の貢献は次に示すとおりである。

- ソースコードの高速化を題材とし、言語モデルの高速化能力を定量的かつ多面的に評価できる手法を提案した
- ソースコードの高速化能力において、SLMがLLMに比べ高速化能力は劣るがコードを壊しにくい長所を発揮する事例を発見した
- SLMとLLMを組み合わせるとLLMのみと比較して高速化能力を向上できることを発見した

2　関連研究

LLMが生成したコードを実行して評価する代表的なベンチマークとしてHumanEval [7] やMBPP [8] がある。これらはタスクの多様性・コード実行の安全性・過剰適合などの問題が指摘されており、xCodeEval [9], SAFIM[4] やLiveCodeBench [10] など新しいベンチマークが提案されている。特にxCodeEvalは競技プログラミングコンテストサイトであるCodeforces[1)]をもとにしており、自然言語とコードの相互変換や間違ったコードの修正など様々なタスクからなる大規模なデータセットと、その評価のためのAPIサーバーが含まれている。

3　実験の概要

実験は5つの段階からなる手順で実施した。本章ではそれらを順に説明する。

3.1　データセット作成

本研究では、2章でも取り上げたxCodeEvalのデータセットを元にした。ただし、この中には、今

1）https://codeforces.com/

表 1: フィルタリング後（サンプリング前）のデータ数。難易度については、Codeforces にて示されている点数を、A(1000 点以下)、B(1100-1400 点)、C(1500-1800 点)、D(1900 点以上) と区分した

言語＼難易度	A	B	C	D	合計
C	224	214	92	70	600
C#	3505	1574	429	72	5580
C++	325	255	924	2045	3649
Go	42	20	6	5	73
Javascript	290	21	7	3	321
PHP	445	96	19	4	564
Python	3617	2178	1856	591	8242
Ruby	1421	399	98	25	1943
Rust	31	16	13	22	82
合計	9900	4773	3444	2837	20954

回利用したいコード・テキスト・テストの 3 点がすべて揃っていないものや、テストを通過しないコードが含まれている。また、Java のように xCodeEval の評価フレームワークでは実行時間が計測できないプログラミング言語のコードも含まれている。そのためまず、このような本研究に用いることができないデータをフィルタリングした。このフィルタリングによって、実験に使用できるデータの総数は 20,954 件となった。

フィルタリング後の内訳を表 1 に示す。これを見ると、プログラミング言語や難易度に大きな偏りがあることがわかる。そこで本実験ではそのような偏りを排除するために、言語ごと・難易度ごとに一様乱択で 25 件ずつサンプリングした。この条件を満たすコード数が 25 に満たない場合も発生したが、そのような場合には全数を 1 回ずつ使用することとした。この結果、実際に使用したサンプル数は 688 件となった。

3.2 LLM によるコード生成

vLLM[11] ライブラリのバッチ推論機能を使用して、各モデルに対してコードを入力し、プロンプトを与えたうえで返答を収集した。

モデルは次の 4 種類を評価対象とした。

- Llama3.3-70B: meta-llama/Llama-3.3-70B-Instruct
- Llama3.2-1B: meta-llama/Llama-3.2-1B-Instruct
- Qwen2.5-32B: Qwen/Qwen2.5-Coder-32B-Instruct
- Qwen2.5-1.5B: Qwen/Qwen2.5-Coder-1.5B-Instruct

Qwen 2.5 はプログラミング向けの追加学習がされている特化型モデルで、 Llama 3 はより大規模なデータセットで学習された汎用モデルである。それぞれ、 LLM に該当するサイズとして 32B と 70B 、SLM に該当するサイズとして 1.5B と 1B を用いた。

システムプロンプトは次の 4 種類を使用した。

- `rewrite` : 「コードを書き換えて」という指示だけを与え、高速化を指示しない
- `simple` : 「コードを高速化して」という指示だけを与える
- `general` : `simple` に一般的な高速化手法（SIMD、メモリアクセス最適化など）をまとめたテキストを加える
- `competitive` : `simple` に競技プログラミングでよく使われる高速化アルゴリズム（二分探索、ダイクストラ法、DP など）をまとめたテキストを加える

`general` と `competitive` プロンプトについては、各手法の名称と概要のみを自然言語で記載することで、各プログラミング言語についてどのように実装するかについて、モデルが元々持っている知識と能力が発揮されることを期待した。システムプロンプトの詳細と具体例は付録 B を参照されたい。

ユーザープロンプトは問題ごとにテンプレートとして与えた。実際の内容は付録 C のとおりである。

コードの正しさを維持しながらバリエーションのある結果を得るために、温度パラメータは 0.3、Top-p は 0.95 を設定した。また各設定でシードを変更して複数回生成させることで、同じ問題に対して複数の異なる生成結果を得た。その他の設定は付録 D に記載した。

3.3 コード抽出

LLM の markdown 記法の出力から、コードブロックに相当するバッククォート 3 つで囲まれた最初のブロックを取り出す後処理を実施した。その結果、目視確認の範囲においては多くのコードが正常に抽出できていることが確認された。抽出できなかったものは次の段階において正答できなかったものとして判定される。

3.4 実行時間測定

ここでは最初に、 xCodeEval API に抽出したコードを与え、各テストケースに正答するかを評価し

た。その後、正答した場合には、その標準入力を与えてコンパイル済みのバイナリを呼び出した時点から、解答を標準出力して実行が終了した時点の処理時間を「実行時間」とし、ケースごとに 1 回ずつ測定した。API による実行時間測定は `/proc/{pid}/stat` を利用して行われており、分解能は 0.01 秒である。

3.5 実行時間比較

各問題について（LLM が出力したコードを使った場合の各テストケースの実行時間の合計）÷（データセットの既存コードを使った場合の各テストケースの実行時間の合計）を相対実行時間として収集した。その後、得られた相対実行時間の累積分布を描画して、モデルの高速化能力を比較した。

4 予備実験

モデルごとの性能評価をする本実験の前に、本実験の比較で用いるプロンプトと推論回数を予備実験にて検討した。

4.1 プロンプトによる影響

図 1a に、モデルを Qwen2.5-32B に固定してプロンプトを変えた場合に得られた相対実行時間を示す。実行可能なコード数は、高速化の指示を含まない `rewrite` プロンプトが他の 3 種類のプロンプトに比べて顕著に多くなった。一方で、相対実行時間が 0.5 未満の（つまり 2 倍を超える高速化がされた）コードの数は、高速化の指示を含むプロンプトよりわずかに減少する結果となった。また、他の 3 種類は全体として同様の傾向であり、プロンプトによる影響はあまり大きくなかった。この結果から、本研究のように高速化する能力に重点を置いて評価する場合には、高速化指示を含むほうが良いが事例の具体指示は必要ないことが分かった。そこで今後の実験では `simple` プロンプトを使用して評価することとした。

4.2 推論回数による影響

プロンプトを `simple` に固定して、Qwen2.5-32B モデルで推論回数を 1,2,5,10 回に変更した時に最短実行時間を選択した場合の結果を図 1b に示す。この図から、試行回数を増やしても、高速化の成功率が一様に向上するだけで全体的な傾向は大きく変化しないことが分かった。そのため今後の実験では 1 回ずつ推論した結果でも比較には十分と判断し、1 回の結果のみを用いた。

5 本実験の結果と考察

5.1 モデルの比較

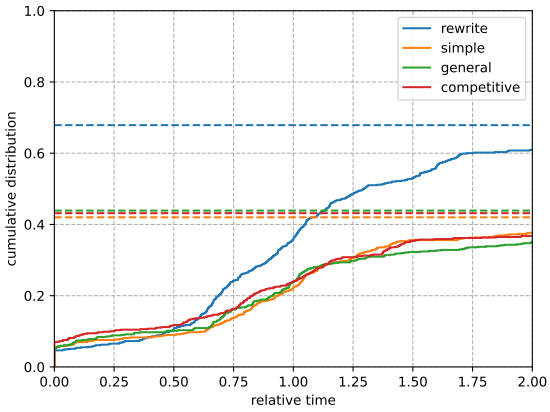
モデルごとに `simple` プロンプトで 1 回実行したときの相対実行時間の比較を図 1c に示す。実行可能なコードを返す割合が最も高かったのは SLM である Qwen2.5-1.5B であった。一方で、高速化されるコードについては、相対実行時間が 0.5 倍未満になる（2 倍を超えて高速化される）のは LLM である Llama3.3-70B および Qwen2.5-32B が高くなった。このことから、SLM は LLM に比べて高度な高速化を行う能力は低い一方で、コードを壊してしまうようなリスクは低い特性を持つことが示唆された。

5.2 複数モデルの利用

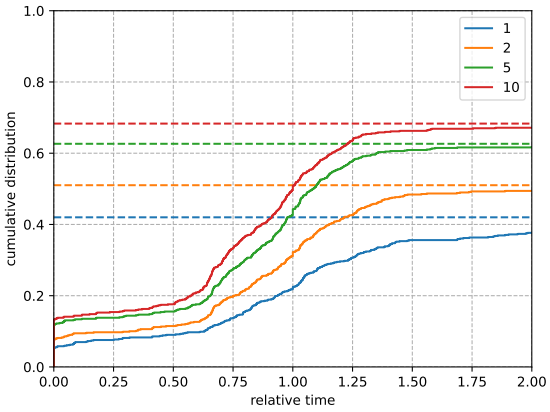
前節の結果によってモデルによって長所特性が異なることが分かったが、実用を考えると、その異なる適性の両方をもった結果を得たい。そのような目的のためには複数モデルを組み合わせることが考えられるが、その時に適性がどのように変化するかを比較した。比較結果を図 1d に示す。この図から、Qwen2.5-32B の性能を向上させるには、他の LLM である Llama3.3-70B と組み合わせたり、Qwen2.5-32B 自身を 2 回推論させるより、SLM である Qwen2.5-1.5B を組み合わせたほうが高い性能が得られていることが確認された。このことから、複数のモデルを組み合わせる場合においても、LLM だけでなく SLM を用いて組み合わせるほうが、コードを壊しにくい SLM の利点と、高い性能を出しやすい LLM の利点を組み合わせることができ、計算量を節約して良い結果を得られることが示唆された。

5.3 主要ベンチマークとの比較

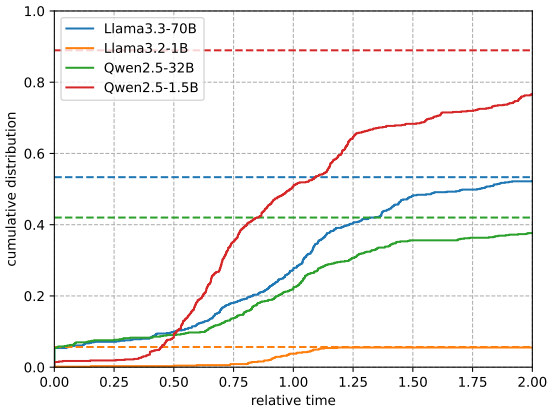
相対実行時間 1 倍、 0.5 倍を達成したコードの百分率を、モデルカードとして提供されている主要ベンチマークと比較した結果を表 2 に示す。MMLU[12] など主要ベンチマークは SLM のスコアが LLM の半分程度となっているが、 1 倍のスコアは Qwen2.5-1.5B のほうが LLM より高くなっている。言い換えると、この指標はコードに特化して学習された SLM を LLM より高く評価するという点で特異的である。



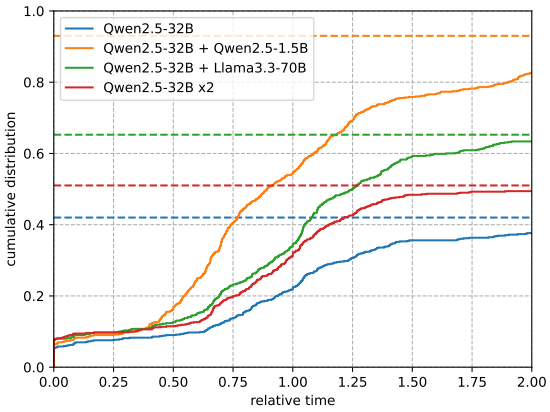
(a) プロンプトの比較



(b) 推論回数の比較



(c) モデルの比較



(d) 複数モデル利用時の比較

図 1: 相対実行時間の実験結果。実線がその相対実行時間未満に短縮改善できた累積割合を示し、破線は実行可能なコードが出力された割合を示す。相対実行時間が 0 になっているのは、各テストケースがすべて時間分解能（0.01 秒）未満まで改善されたことを示す。

表 2: 主要ベンチマークとの比較 (*は Base モデル)

	Llama 3.3-70B	Llama 3.2-1B	Qwen 2.5-32B	Qwen 2.5-1.5B
MMLU	86.0	49.3	77.6	42.0*
MMLU-Pro	68.9	-	62.3	-
IFEval	-	59.5	79.9	-
HumanEval	88.4	-	92.7	70.7
EvalPlus	87.6	-	75.1	59.4
MATH	77.0	30.6	76.4	15.4*
GSM8K	-	44.4	93.0	34.5*
相対実行時間				
1 倍	27.2	3.8	21.8	50.7
0.5 倍	9.9	0.4	9.0	8.6

6 おわりに

この論文では、ソフトウェアの高速化という能力に着目して言語モデルを評価する手法を提案し、実際に SLM と LLM における特性の違いを明らかにした。今後の展開として、ユーザープロンプトの変更によってコードに関連するテキストの情報量が変化する場合の影響や、より多くのモデル・データセットに対する性能比較を実施することで、モデルごとの特性の違いがより明らかになると考えられる。また、今回の評価手法を、より汎用的かつ簡易に利用できる評価ベンチマークとして整備し公開することで、今後も進化が期待される各言語モデルの性能評価の一助となることを期待したい。

謝辞

本研究の実験環境には、株式会社フィックスターズの Fixstars AI Booster クラウドを活用しました。

参考文献

[1] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2024.

[2] Zhenyan Lu, Xiang Li, Dongqi Cai, Rongjie Yi, Fangming Liu, Xiwen Zhang, Nicholas D. Lane, and Mengwei Xu. Small language models: Survey, measurements, and insights, 2024.

[3] Chien Van Nguyen, Xuan Shen, Ryan Aponte, Yu Xia, Samyadeep Basu, Zhengmian Hu, Jian Chen, Mihir Parmar, Sasidhar Kunapuli, Joe Barrow, Junda Wu, Ashish Singh, Yu Wang, Jiuxiang Gu, Franck Dernoncourt, Nesreen K. Ahmed, Nedim Lipka, Ruiyi Zhang, Xiang Chen, Tong Yu, Sungchul Kim, Hanieh Deilamsalehy, Namyong Park, Mike Rimer, Zhehao Zhang, Huanrui Yang, Ryan A. Rossi, and Thien Huu Nguyen. A survey of small language models, 2024.

[4] Linyuan Gong, Sida Wang, Mostafa Elhoushi, and Alvin Cheung. Evaluation of llms on syntax-aware code fill-in-the-middle tasks, 2024.

[5] Continue faq: I want better completions, should i use gpt-4?

[6] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024.

[7] Mark Chen, et al. Evaluating large language models trained on code, 2021.

[8] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

[9] Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. xcodeeval: A large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval, 2023.

[10] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024.

[11] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention, 2023.

[12] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding, 2021.

A ソフトウェア高速化の具体例

まずC言語のループ範囲の省略について例示する。元のコードは下記のようになっていた。

```
for (int x = 0; x <= B; x++) {
    for (int y = 0; y <= C; y++) {
        int rem = N - 2 * x - 4 * y;
        if ((rem >= 0) && (rem <= A)) res
        ++;
```

Qwen2.5-32Bによる出力では、あらかじめ条件を満たさない範囲をループから省略することで、実行時間が0.17秒から0.01秒に高速化された。

```
for (int x = 0; x <= B && 2 * x <= N; x++)
 {
    for (int y = 0; y <= C && 4 * y <= N -
     2 * x; y++) {
        int rem = N - 2 * x - 4 * y;
        if ((rem >= 0) && (rem <= A)) res
        ++;
```

別の例として、C++言語の `std::unordered_map` を使用することによる高速化についても挙げる。問題は $_{i+1} = (ar_i + b) \mod m$ で生成される数列の周期を求めるもので、元のコードは十分な長さ生成してから同じ値が出てくるまで後ろ向きに走査するというものだった。

```
for(i = 1; i < 50000001l; ++i)
    r[i] = (a * r[i - 1] + b) % m;
for(int j = i - 2; j >= 0; --j)
    if(r[i-1] == r[j]) {
        cout << i - j - 1;
        return 0;
    }
```

Qwen2.5-32Bはより単純に前向きに走査するコードを出力し、3.32秒から0.08秒への高速化を達成した。

```
unordered_map<int, int> seen;
while (true) {
    if (seen.find(current) != seen.end())
    {
        cout << step - seen[current];
        return 0;
    }
    seen[current] = step;
    current = (a * current + b) % m;
    step++;
}
```

B システムプロンプトの例

作成したシステムプロンプトは、モデルにとって解釈がしやすいことを考えて、いずれも英語で記載した。また、後処理のしやすさを考えて、各プロンプトにはmarkdown形式で出力するように指示を加えた。作成した4種類のプロンプトのうち `competitive` プロンプトの例を次に示す。これはLlama3.1-70Bに生成させたのち手動で修正したテキストである。`simple` プロンプトは、`competitive` プロンプトの第1パラグラフのみ使用したものである。

```
You will be given a problem statement for
the programming contest and a code that
answers it. Please modify this code to
make it faster. Please output the result
as a code embedding in markdown.

Some of the algorithms often used in
competitive programming are:

- Binary Search: A search algorithm that
uses a sorted array or list to determine
whether it contains a certain value.
- Depth-First Search (DFS) and Breadth-
First Search (BFS): Algorithms for
searching graphs and tree structures. DFS
goes as deep as possible, while BFS
prioritizes searching adjacent nodes.
- Dijkstra's Algorithm: An algorithm for
finding the shortest path in a graph. It
supports weighted graphs.
- Dynamic Programming: An algorithm that
breaks down a complex problem into smaller
 subproblems and combines the solutions to
 find the overall solution.
- Greedy Algorithm: An algorithm that
finds the overall optimal solution by
making the optimal choice at each step.
- Backtracking: An algorithm that searches
 recursively until a solution is found,
and returns to the previous step if it
fails.
- Euclidean algorithm: An algorithm for
finding the greatest common divisor (GCD)
of two numbers.
- Fermat's Little Theorem: An algorithm
for efficiently calculating the remainder
of large numbers.
- FFT (Fast Fourier Transform): A fast
Fourier transform algorithm. It is used
for polynomial multiplication and
convolution.
- Segment Tree: A data structure for
efficiently calculating the median and
range sum of arrays and lists.
- Union-Find (UF): A data structure for
managing sets and determining whether two
elements belong to the same set.

These algorithms are frequently used to
solve competitive programming problems,
and understanding the algorithms and the
ability to implement them are important.
```

C ユーザープロンプト

Jinja言語を用いたmarkdownテンプレート形式で記載し、問題に応じて自動で適用されるようになっている。

```
# problem

{{ description }}

# answer

```{{ language }}
{{ code }}
```
```

D 生成時の環境設定

NVIDIA H100 SXMが8枚搭載されたサーバーを使用し、Llama3.2-1B, Qwen2.5-1.5Bモデルは Tensor Parallel size (TP)=1, Qwen2.5-32B は TP=2, Llama3.3-70B は TP=4 と設定した。モデルの量子化は適用せず、コンテキスト長はモデルのデフォルトを使用した。最大出力トークン数は4096とした。