Avaliação de Modelos de Linguagem de Grande Escala na Geração Automática de Código Paralelo OpenMP

Julio Cesar Nogueira[1], **André L. S. Kawamoto**[1], **João Fabrício Filho**[1], **Rogério A. Gonçalves**[1]

[1]Programa de Pós-Graduação em Ciência da Computação (PPGCC-CM)
Departamento Acadêmico de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR), campus Campo Mourão
Via Rosalina Maria dos Santos, 1233 - Vila Carolo – 87301-899
Campo Mourão – PR – Brasil

`julionogueira@alunos.utfpr.edu.br, {kawamoto,joaof,rogerioag}@utfpr.edu.br`

Resumo. *Este trabalho apresenta uma avaliação de Modelos de Linguagem de Grande Escala (LLMs) — incluindo ChatGPT 4.5, Gemini 3, Qwen, CodeLlama, entre outros — na geração/anotação automática de código paralelo OpenMP. Utilizando a suíte de benchmarks PolyBench/C, comparou-se o desempenho das versões geradas por Inteligência Artificial contra implementações manuais de especialistas e ferramentas de compilação determinísticas, como o LLVM PoLLy. Os resultados experimentais revelam que LLMs modernos produzem código com speedup competitivo, superando ferramentas tradicionais em cenários cache-friendly, embora ainda enfrentem desafios em kernels com dependências de dados complexas nos quais o modelo poliédrico prevalece.*

1. Introdução

Aplicações de Computação de Alto Desempenho (HPC) exigem cada vez mais o uso eficiente de arquiteturas *multicore* e *manycore* para aproveitar o poder de processamento paralelo disponível. O padrão `OpenMP` [Dagum and Menon 1998] destaca-se nesse cenário, permitindo a paralelização de código via *diretivas de compilação (pragmas)* sem a necessidade de reescrever as aplicações por completo. Contudo, a inserção manual dessas diretivas pode ser uma tarefa complexa e altamente suscetível a erros. Exigindo ajustes especializados por parte do desenvolvedor, que precisa analisar as dependências de dados, definir políticas de escalonamento (*scheduling*) e gerenciar a localidade de memória. Por não eximir o programador desse profundo trabalho analítico de otimização, a anotação manual torna-se um processo demorado e não trivial.

Ferramentas de compilação *source-to-source*, como o `LLVM PoLLy` [GROSSER et al. 2012] e o `DawnCC` [Mendonça et al. 2017], utilizam modelos matemáticos (como o Modelo Poliédrico e Análise de Intervalo Simbólico) para garantir a corretude da paralelização. No entanto, essas abordagens tendem a ser conservadoras: se não for possível provar matematicamente a independência dos dados, a paralelização é abortada.

Recentemente, *Modelos de Linguagem de Grande Escala* (LLMs) emergiram como uma alternativa probabilística promissora [Rozière et al. 2024]. No entanto, faltam avaliações sistemáticas comparando múltiplos LLMs do estado da arte com *benchmarks* de HPC. Este trabalho visa preencher essa lacuna, avaliando a eficácia de sete LLMs na inserção automática de diretivas `OpenMP`, comparando com código gerado por ferramentas determinísticas e especialistas humanos.

2. Metodologia

A metodologia compara três paradigmas: *(i) Humano Especialista (Versão Sequencial/-Versão OpenMP do PolyBench-ACC)*, *(ii) Automático Determinístico (PoLLy, DawnCC)* e *(iii) Automático Probabilístico (LLMs)*.

Foram selecionados cinco *kernels* da suíte `PolyBench/C` versão 4.2.1 [Pouchet and Yuki 2012]: `2mm`, `3mm`, `gemm`, `atax` e `syr2k`. Os três primeiros representam padrões fundamentais de multiplicação de matrizes, amplamente utilizados em aplicações de IA e HPC, enquanto `atax` e `syr2k` foram escolhidos por apresentarem acessos assimétricos e dependências de dados mais complexas, o que desafia a capacidade de inferência semântica dos modelos. Os códigos gerados por cada modelo, bem como os artefatos comparativos, foram disponibilizados publicamente para reprodutibilidade[1]. Para geração via LLM, utilizou-se engenharia de *prompt* iterativa nos modelos: *CodeLlama*, *Codestral*, *DeepSeek Coder*, *Granite Code*, *Gemini 3*, *Qwen* e *ChatGPT 4.5*.

Para evitar o alto custo computacional do treinamento adicional (*fine-tuning*) e extrair o conhecimento intrínseco aos parâmetros dos modelos, adotou-se a abordagem de *Zero-Shot Prompting*. O objetivo foi forçar a LLM a se comportar de maneira semelhante a um compilador *source-to-source*, aplicando otimizações estritas sem alterar a lógica original do programa. Para mitigar alucinações na geração de código e garantir um controle do experimento, foi empregado um *prompt* padronizado descrevendo a tarefa a ser realizada *(Task)*, objetivo *(Goal)*, restrições sobre alterações *(Strict Rules)*, instruções, formato de saída, juntamente com o código a ser analisado.

Para assegurar a consistência na condução do experimento, todo o fluxo de compilação e execução foi orquestrado por *scripts* em Shell e Python. Durante a execução, foram utilizadas as ferramentas do GNU/Linux para a captura do *tempo de execução* e a *utilização de CPU*. Os dados foram consolidados e analisados utilizando as bibliotecas `pandas` e `matplotlib` do `Python`, permitindo o cálculo do *Speedup* e da Eficiência Paralela.

Além da avaliação de desempenho (quantitativa), estabeleceu-se uma análise do código gerado para contornar descrições vagas. Para evitar classificar o código apenas como *"sintaticamente correto"*, a avaliação classificou a maturidade das soluções geradas pela IA nos seguintes critérios específicos:

- **Corretude Sintática e Semântica:** Define se o código gerado é compilável, se as diretivas (*pragmas*) são sintaticamente bem formadas (como o uso válido de `section`, `reduction` e `parallel for`) e se a lógica mantém os resultados numéricos corretos do algoritmo original.
- **Robustez:** Avalia o uso adequado e seguro do escopo de dados (cláusulas `private`, `shared`) e mecanismos de sincronização para a prevenção de condições de corrida (*data races*).
- **Boas Práticas e Completude:** Analisa a legibilidade e se o modelo aplicou estratégias algorítmicas eficientes, como o uso apropriado das cláusulas `schedule` e `collapse` para o balanceamento de carga.

[1] Repositório de Artefatos: `https://github.com/nogueirj/LLM-Optimisation-PPGCC-CM/`

- **Uso de *Features* Avançadas:** Verifica se a IA teve a iniciativa de utilizar construtores mais modernos da especificação OpenMP (como `task, taskloop` para cargas irregulares, `simd` para vetorização ou operações `atomic`).

Essa distinção permitiu avaliar não apenas a capacidade das ferramentas e dos LLMs de gerar código funcional, mas também o quão aderentes estão às melhores práticas da Computação de Alto Desempenho.

3. Experimentos e Resultados

Os experimentos foram executados em um nó com processador **AMD Ryzen Threadripper 7970X** (32 núcleos, 64 threads) e 256 GB de RAM DDR5. O compilador utilizado foi o `GCC` na versão 13.3 com a *flag* `-O3 -fopenmp`. Métricas de *speedup* e eficiência paralela foram coletadas (média de 10 execuções). Os resultados foram divididos entre cenários `Standard` (dados cabem na cache) e `Large` (limitados pela memória RAM).

O desempenho foi analisado no conjunto de *datasets* admitidos pelo `Polybench/C`. Um comparativo do desempenho dos códigos gerados pelos modelos LLMs com a versão do código OpenMP anotado manualmente é apresentado na Tabela 1. Os modelos LLMs demonstraram desempenho superior para o *dataset* `Standard`, o **ChatGPT 4.5** liderou com *speedup* médio de **38,77x**, seguido pelo *Gemini* (35,56x), superando a versão manual (31,98x) e a versão gerada pelo `PoLLy` (18,24x).

Tabela 1. Ranking de Desempenho - Dataset STANDARD (64 Threads)

Modelo	Tempo Médio (s)	Speedup	Eficiência
ChatGPT 4.5	0.0811	38.77x	0.61
Gemini 3	0.0885	35.56x	0.56
OpenMP (Manual)	0.0919	31.98x	0.50
Codestral	0.0960	30.06x	0.47
Qwen	0.1005	28.96x	0.45
PoLLy	0.1650	18.24x	0.28
CodeLlama	1.3574	11.05x	0.17

A escalabilidade (visualizada na Figura 1) mostra que modelos como *ChatGPT* e *Gemini* atingem crescimento quase linear até 32 threads, evidenciando estratégias eficientes de escalonamento (*scheduling*), enquanto o *CodeLlama* satura precocemente devido a paralelizações ingênuas.

Analisando os resultados para o *benchmark* `syr2k` nos três paradigmas estipulados na Seção2, temos que para o *dataset Large*, a otimização manual retomou a liderança geral (23,98x). Um resultado crítico ocorreu no *kernel* `syr2k`. Enquanto LLMs e a versão manual saturaram, a ferramenta `PoLLy` atingiu um *speedup* de **59,32x**. Isso ocorre porque o **PoLLy** aplica transformações poliédricas (*tiling* avançado) que maximizam a localidade de dados, algo que a *"intuição semântica"* dos LLMs não replicou com eficácia. Por outro lado, no `gemm` (Standard), o **ChatGPT** atingiu 51,81x, superando largamente ferramentas tradicionais.

Todos os LLMs geraram código sintaticamente correto. **ChatGPT** e **Gemini** destacaram-se pelo uso de cláusulas como `collapse` e `schedule(static)`. Contudo, nenhum modelo sugeriu construtores modernos como `task, taskloop` ou `simd` espontaneamente. O modelo **Qwen** destacou-se como uma opção *open weights* altamente competitiva, rivalizando com modelos proprietários.

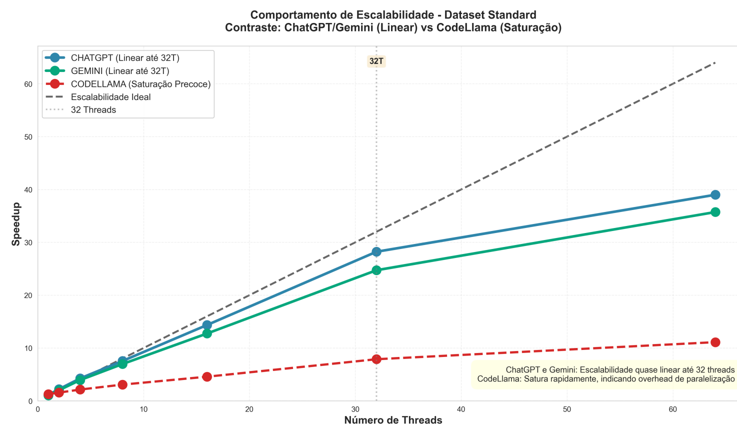

Figura 1. Curvas de escalabilidade no Dataset Standard

4. Conclusão

Este trabalho analisou empiricamente a maturidade de modelos de LLMs para gerar código `OpenMP` de qualidade. Aplicou-se Engenharia de *Prompt Zero-Shot* com alguns códigos da suíte `Polybench/C`. O código gerado pelos LLMs foi comparado com abordagens que utilizam a inserção de diretivas manual e por ferramentas de compilação. Modelos de código aberto como o **Qwen** mostraram-se alternativas promissoras localmente. No entanto, para *kernels* exigindo transformações em domínios de memória complexos (como `syr2k`), a exatidão das ferramentas que utilizam o modelo poliédrico, como o **PoLLy** mostra-se superior as capacidades probabilísticas dos modelos atuais.

Como trabalhos futuros, pretende-se expandir a avaliação para o restante da suíte `PolyBench` e outros *benchmarks* consolidados, como `SPEC CPU` e `PARSEC`. Além disso, planeja-se investigar o impacto de estratégias de *few-shot prompting* e modelos emergentes de raciocínio, bem como aprofundar a análise estatística dos resultados e o uso de recursos avançados do padrão `OpenMP`.

Referências

Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

GROSSER, T., GROESSLINGER, A., and LENGAUER, C. (2012). Polly — performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(04):1250010.

Mendonça, G., Guimarães, B., Alves, P., Pereira, M., Araújo, G., and Pereira, F. M. Q. (2017). DawnCC: Automatic Annotation for Data Parallelism and Offloading. *ACM Trans. Archit. Code Optim.*, 14(2).

Pouchet, L.-N. and Yuki, T. (2012). PolyBench/C: The polyhedral benchmark suite. `http://polybench.sourceforge.net`. Versão 4.2.1.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. (2024). Code llama: Open foundation models for code.