Rust como Engine para Transformers: FFI Estável, Benchmarks e Controle de Dispositivo

Daniel Pontes[1], **Murilo Salem**[1], **Marcos Alves**[1]
Karen Satie[1], **Gerson Geraldo H. Cavalheiro**[1]

[1]Centro de Desenvolvimento Tecnológico
Universidade Federal de Pelotas
96.010-610 – Pelotas – RS – Brasil

{dhspbarretos,mcsalem,mlalves,ks.ono,gersonc}@inf.ufpel.edu.br

Resumo. *Este artigo apresenta a engine Rust usada pelo projeto* `neural-lm-hpc` *para executar workloads Transformer. A engine concentra armazenamento tensorial, modelo, tokenizer, otimizador, exports FFI e backends CPU/CUDA, enquanto scripts reprodutíveis de benchmark reutilizam um runner de profiling simples. As principais contribuições sao um runtime com controle explícito de dispositivo, uma C ABI estável consumida pelo Go e scripts operacionais para throughput, latência e memória. Os resultados em CPU-only mostram o custo esperado de escalar de 125M para 1.3B parâmetros e estabelecem uma baseline reprodutível para futuras medições em CUDA.*

1. Introdução

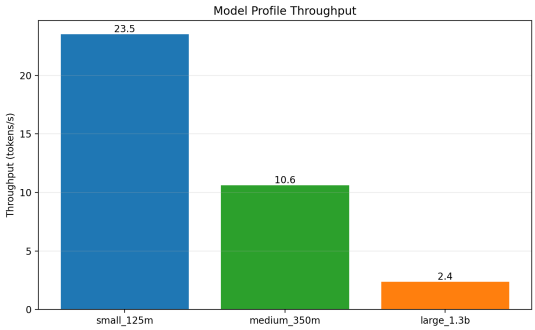
Transformers modernos combinam arquiteturas expressivas com uma demanda severa por *throughput*, controle de memoria e reprodutibilidade experimental [Vaswani et al. 2017, Shoeybi et al. 2019, Brown et al. 2020]. Em projetos de pesquisa, a camada numérica também precisa conviver com FFI (*Foreign Function Interface*), testes e perfis CPU-only para CI (*Continuous Integration*), sem perder a possibilidade de explorar aceleração por GPU [Rasley et al. 2020, Dao et al. 2022].

No `neural-lm-hpc`, a escolha por Rust busca equilibrar esses requisitos. A linguagem permite um núcleo com *ownership* explicito, boa ergonomia para FFI e modularização clara entre tensores, modelo, tokenizer, otimizador e código CUDA. O desenho do modelo segue escolhas frequentes em LLMs contemporâneos, como RoPE e RMSNorm [Su et al. 2021, Zhang and Sennrich 2019, Touvron et al. 2023]. Este artigo resume o desenho da *engine* e os resultados operacionais já coletados no repositório.

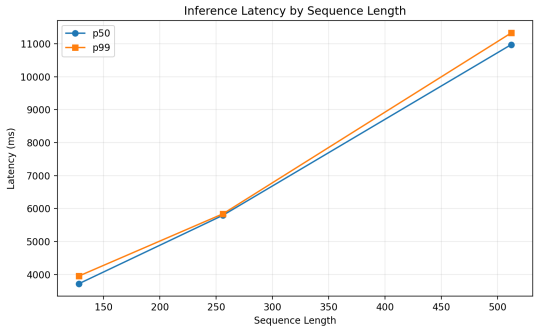
2. Desenho da Engine Rust

A *engine* organiza seu código em módulos dedicados: `tensor/` concentra `Shape`, `Storage` e operações; `model/` implementa o *Transformer*; `optim/` cobre AdamW, *scheduler*, *clipping* e *checkpointing*; `tokenizer/` fornece BPE e *dispatch* SIMD; `cuda/` encapsula *device registry*, *streams*, *wrappers* de cuBLAS e lançadores FFI. O mesmo crate expõe uma C ABI consumida pelo orquestrador Go para inicialização, treino, inferência, tokenização e leitura de métricas.

Duas decisões recentes foram relevantes. A primeira foi remover o hardcode do dispositivo CUDA, preservando o `device_id` escolhido em `engine_init` e



(a) Throughput em CPU-only por configuração.



(b) Latência do preset `small_125m`.

Figura 1. Benchmarks reprodutíveis gerados pelos scripts do repositório a partir do runner `model_profile`.

reutilizando-o na criação do modelo, na inferência e na coleta NVML. A segunda foi ampliar o *runner* `model_profile` com `--samples-json`, permitindo que o script de latência calcule percentis sem depender de ferramentas externas. Essas duas mudanças reforçam a tese central do trabalho: previsibilidade operacional vale tanto quanto velocidade bruta, especialmente em uma *stack* que pretende evoluir para kernels especializados e caminhos de atenção mais eficientes [Dao et al. 2022].

3. Resultados Operacionais

A Figura 1 resume duas perspectivas complementares. Em *throughput*, a baseline CPU-only com `seq_len=64` e `batch=1` produziu 23,5 tok/s no `small_125m`, 10,6 tok/s no `medium_350m` e 2,37 tok/s no `large_1.3b`. Em latência, o preset `small_125m` apresentou *p50* de 3,72 s para sequencias de 128 tokens, 5,80 s para 256 tokens e 10,97 s para 512 tokens. Embora esses números sejam modestos, eles servem como *baseline* determinística e comparável para iterações futuras em CUDA.

Tabela 1. Resumo operacional do caminho CPU-only em `seq_len=64`.

Modelo	Throughput (tok/s)	RSS pico (MB)	Latência média (ms)
`small_125m`	23,5	598	2722
`medium_350m`	10,6	1420	6027
`large_1.3b`	2,37	6252	27052

O perfil de memoria mostra o mesmo comportamento esperado de crescimento por escala, saindo de aproximadamente 598 MB no `small_125m` para 6,25 GB no `large_1.3b`. Mais importante do que o valor absoluto foi a reprodutibilidade da coleta: *throughput*, latência e memoria são gerados por scripts versionados, com saída JSON e possibilidade de *plot* automatizado.

4. Discussão

Rust mostrou-se adequado como camada numérica por três motivos. Primeiro, a linguagem favorece *ownership* explicito e evita *aliasing* acidental em estruturas que misturam tensores, buffers e estado de dispositivo. Segundo, a C ABI permanece pequena

e estável o bastante para ser consumida por Go sem replicar a complexidade interna. Terceiro, o *runner* de *benchmark* permite ligar instrumentação adicional sem poluir a API publica. Esse desenho se aproxima de uma necessidade recorrente em stacks de treinamento e inferência de larga escala: manter o caminho critico enxuto e observável [Shoeybi et al. 2019, Rasley et al. 2020, Kwon et al. 2023].

As limitações atuais ainda importam. Os resultados apresentados aqui são CPU-only; a campanha final em CUDA e o comparativo sistemático com PyTorch ficaram fora do escopo desta versao curta. Da mesma forma, perfis finos com Nsight e explorações de kernels especializados exigem um artigo próprio.

5. Conclusão

O estado atual da *engine* Rust confirma que a separação entre controle em Go e execução numérica em Rust e tecnicamente viável. A *engine* já entrega FFI estável, controle explicito de dispositivo, benchmark reprodutível e uma baseline coerente para evolução rumo a execução acelerada em GPU e campanhas comparativas mais amplas.

Referências

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020). Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. (2022). Flashattention: Fast and memory-efficient exact attention with io-awareness.

Kwon, W., Lee, Z., Li, S., Zhuang, Y., Sheng, Y., Zheng, L., Yu, C., Gonzalez, J. E., Zhang, H., and Stoica, I. (2023). Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, pages 611–626. ACM.

Rasley, J., Rajbhandari, S., Ruwase, O., and He, Y. (2020). DeepSpeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 4739–4740. ACM.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. (2019). Megatron-lm: Training multi-billion parameter language models using model parallelism.

Su, J., Lu, Y., Pan, S., Wen, B., and Liu, Y. (2021). Roformer: Enhanced transformer with rotary position embedding.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. (2023). Llama 2: Open foundation and fine-tuned chat models.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pages 5998–6008. Curran Associates, Inc.

Zhang, B. and Sennrich, R. (2019). Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32.