Go como Orquestrador de Treinamento e *Serving* para Cargas de Trabalho *Transformer*

Daniel Pontes[1], **Murilo Salem**[1], **Marcos Alves**[1]
Karen Ono[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 descreve o estado operacional atual de um orquestrador em Go para workloads Transformer no projeto* `neural-lm-hpc`*. O Go concentra configuração, controle de treino, checkpointing, serving gRPC e observabilidade Prometheus, enquanto a execução numérica permanece contida em uma engine Rust. A principal contribuição é uma camada de orquestração com baixo acoplamento, que preserva a configuração original do modelo, isola a fronteira FFI (Foreign Function Interface) através de interfaces pequenas e torna o serving testável de forma independente da engine numérica. O resultado é uma base prática para ciclos curtos de experimentação.*

1. Introdução

Modelos de linguagem baseados na arquitetura *Transformer* [Vaswani et al. 2017] tornaram-se o paradigma dominante no processamento de linguagem natural, impulsionados por trabalhos de grande escala como o GPT-3 [Brown et al. 2020]. O treinamento e a inferência desses modelos exigem não apenas alto desempenho numérico, mas também uma camada de controle capaz de gerenciar configuração, ciclo de vida, observabilidade e interfaces remotas [Narayanan et al. 2021]. Em sistemas de pesquisa e *serving* de modelos, essa camada de controle muda com frequência maior do que o núcleo tensorial, criando a necessidade de separar o plano de controle do plano de computação [Rasley et al. 2020, Kwon et al. 2023].

O projeto `neural-lm-hpc` foi arquitetado com o objetivo de construir uma infraestrutura de modelos de linguagem de alto desempenho utilizando linguagens de sistemas modernas, sem depender de *frameworks* de aprendizado de máquina baseados em Python. O projeto adota uma separação explícita de responsabilidades: Go coordena treino e *serving*, enquanto Rust concentra tensores, o modelo e os caminhos de execução em CPU e GPU via CUDA. Este artigo discute o módulo orquestrador Go, com ênfase nas decisões que permitiram reduzir o acoplamento com a *engine*, melhorar a testabilidade do servidor gRPC e padronizar a exportação de métricas operacionais. As seções seguintes apresentam os conceitos fundamentais envolvidos, a arquitetura do orquestrador, os mecanismos de validação adotados, uma discussão sobre ganhos e limitações observadas, e os trabalhos futuros planejados.

2. Conceitos Fundamentais

O projeto `neural-lm-hpc` combina três tecnologias centrais, cada uma com papel distinto na arquitetura. A arquitetura *Transformer* [Vaswani et al. 2017], base dos mo-

delos de linguagem modernos como a família GPT [Brown et al. 2020], introduziu o mecanismo de atenção multi-cabeça, que permite alto paralelismo durante o treinamento ao eliminar a dependência sequencial entre posições. A escala desses modelos impõe demandas significativas de memória e largura de banda computacional, motivando otimizações como FlashAttention [Dao et al. 2022] e paralelismo de modelo [Narayanan et al. 2021, Shoeybi et al. 2019] — contexto em que infraestruturas de treinamento eficientes se tornam essenciais. Go atua como plano de controle do projeto: linguagem compilada e estaticamente tipada, com suporte nativo à concorrência via gorrotinas (*goroutines*), oferece integração direta com gRPC e Prometheus sem impor ao motor numérico responsabilidades operacionais. Rust, por sua vez, concentra a execução numérica: suas garantias de segurança de memória em tempo de compilação, sem coletor de lixo, viabilizam acesso eficiente a CUDA e operações de baixo nível. A comunicação entre os dois módulos ocorre via CGo, mecanismo que permite ao código Go chamar funções de bibliotecas Rust exportadas com ABI C, introduzindo uma sobrecarga por chamada discutida na Seção 5.

3. Projeto do Orquestrador

O orquestrador expõe duas entradas principais: `cmd/train` e `cmd/serve`. No caminho de treino, a configuração YAML é carregada uma única vez e seu caminho absoluto original é preservado no campo `SourcePath`. Isso evita divergências entre o perfil escolhido pelo usuário e o perfil de fato instanciado na *engine*, eliminando acoplamento indevido a perfis predefinidos no código.

O caminho de treino encadeia os pacotes `config`, `scheduler`, `pipeline` e `bridge`. O scheduler alimenta a *engine* Rust via CGo, consolidando métricas como vazão de *tokens*, perda de treino, norma do gradiente, taxa de aprendizado e duração de passo, além de disparar o salvamento periódico de pontos de controle. O caminho de *serving* reutiliza a mesma configuração do modelo e publica duas chamadas de procedimento remoto simples — `Health` e `Generate` —, suficientes para testes de fumaça, integração com `grpcurl` e validação inicial do fluxo autorregressivo.

Outra decisão central foi ocultar os tipos concretos da FFI por meio de duas interfaces reduzidas: uma para inferência do modelo e outra para tokenização. Com isso, o servidor gRPC passou a ser exercitado por *test doubles*, sem necessidade de carregar a `libengine` em toda execução de teste. Essa escolha também simplificou a contabilização centralizada de falhas por motivo, melhorando a legibilidade do código e a cobertura dos cenários de erro.

4. Observabilidade e Validação

O *endpoint* `/metrics` expõe grupos distintos de métricas no formato Prometheus. Durante o treino, indicadores de valor instantâneo registram `tokens_per_second`, `training_loss`, `grad_norm`, `learning_rate` e `step_duration_seconds`. No plano de hardware, medidores acompanham utilização, memória de vídeo (VRAM) e temperatura da GPU quando esses dados estão disponíveis. Durante o *serving*, contadores de requisições e falhas convivem com um histograma de latência de inferência e um contador de *tokens* gerados.

Essa estrutura foi validada em duas camadas. A primeira é unitária: o pacote

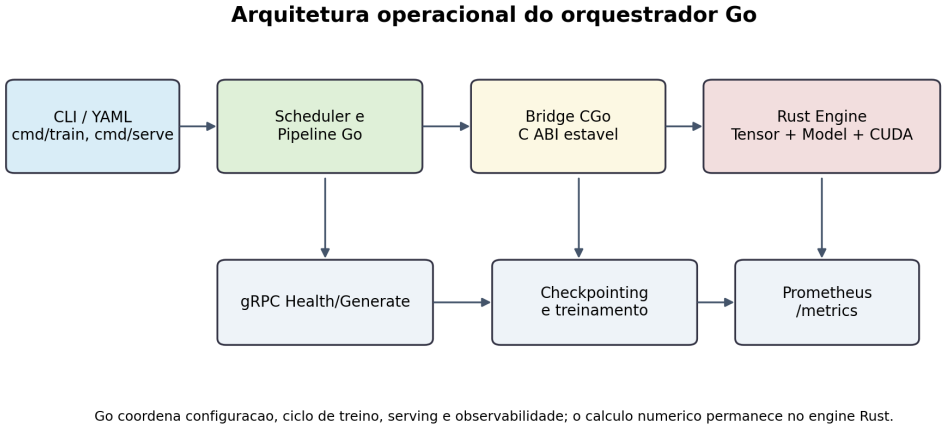

Figura 1. Papel do Go na arquitetura operacional do `neural-lm-hpc`: configuração, treino, *serving* e observabilidade permanecem no plano de controle Go, enquanto a computação numérica segue encapsulada na *engine* Rust.

`internal/server` cobre `Health`, geração bem-sucedida, entrada inválida, falha de tokenização, amostragem e *logits* malformados. A segunda é integrada: `go test ./...` e `make test` executam o caminho Go em conjunto com a suíte Rust. Esse arranjo não mede o acordo de nível de serviço final de produção, mas estabelece uma base reprodutível para as próximas evoluções do projeto, discutidas a seguir.

5. Discussão

O principal ganho pragmático do uso de Go como orquestrador é que a linguagem oferece ferramentas de linha de comando simples, boa integração com gRPC e uma superfície direta para Prometheus, sem forçar a *engine* numérica a absorver responsabilidades operacionais. A fronteira FFI permaneceu estreita e explícita, alinhando-se a uma divisão de responsabilidades semelhante à adotada por sistemas maiores entre controle de experimento, *serving* e execução numérica [Rasley et al. 2020, Kwon et al. 2023].

6. Conclusão

Este trabalho demonstrou que Go é uma escolha efetiva para o plano de controle de cargas de trabalho *Transformer* quando combinado a uma *engine* numérica dedicada em Rust. A implementação atual entrega configuração consistente, salvamento de pontos de controle, *serving* via gRPC, observabilidade com Prometheus e uma base de testes suficiente para sustentar a próxima fase de experimentos do projeto `neural-lm-hpc`.

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.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Korthikanti, V. A., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., Phanishayee, A., et al. (2021). Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, pages 1–15. 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.

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.