Avaliação Energética Sensível ao Regime de Execução em GPU: PyTorch e Pré-processamento Externo em Go/Rust

Murilo Salem[1], **Daniel Pontes**[1], **Luísa Bohm**[1],
Henrique dos Reis[1], **Gerson Geraldo H. Cavalheiro**[1]

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

{mcsalem,dhspbarretos,lhbohn,hdreis,gersonc}@inf.ufpel.edu.br

Resumo. *O custo energético crescente de cargas de IA/ML em GPUs exige benchmarks capazes de medir energia de ponta a ponta, e não apenas throughput. Apresentamos o* ***GC-Bench****, um harness orientado por configuração que expande planos YAML em experimentos reprodutíveis, amostra potência da GPU via NVML com fallback para* `nvidia-smi` *e gera relatórios com base estatística. Avaliamos dois caminhos de execução sobre um mesmo backbone TinyLM em PyTorch: um em PyTorch puro e outro com tokenização externa em Go/Rust. Em 1.018 execuções validadas, o treino apresentou quase empate, enquanto a inferência favoreceu o PyTorch puro, que reduziu o J/token da GPU em 6,73%. Em H2D, memória pinned elevou a largura de banda em 2,82× e reduziu J/GB em 19%. O GC-Bench é disponibilizado como artefato aberto para apoiar estudos reprodutíveis de energia.*

1. Introdução

A pegada energética do treinamento e da inferência de modelos de linguagem tem tornado a eficiência energética uma preocupação central em sistemas. Na prática, escolhas de *runtime* não podem mais ser avaliadas apenas por *throughput* ou latência; elas também precisam considerar potência e custo energético. Essa necessidade é consistente com esforços anteriores em benchmarking e sustentabilidade em aprendizado de máquina, que mostram que avaliações centradas apenas em desempenho são insuficientes para caracterizar cargas modernas de IA [Mattson et al. 2020, Patterson et al. 2021].

Neste trabalho, apresentamos o **GC-Bench**, um *harness* leve para avaliação energética, sensível ao regime de execução, de cargas de IA/ML em GPU. O GC-Bench expande planos experimentais em YAML em matrizes reprodutíveis de execução, amostra energia via NVML com *fallback* para `nvidia-smi` e produz relatórios com base estatística. Avaliamos um caminho em PyTorch puro e um caminho heterogêneo que preserva a mesma computação *TinyLM* em PyTorch, mas adiciona um micropipeline externo de tokenização em Go/Rust. O caminho `go_rust` atual não constitui um backend nativo alternativo, mas sim "PyTorch + pré-processamento externo em Go/Rust". Ao longo dos regimes de treino, inferência e transferência H2D, os resultados mostram que a eficiência depende fortemente do regime de execução, que a inferência favorece o PyTorch puro e que a energia de CPU permanece não desprezível quando medida com RAPL.

2. Metodologia

O GC-Bench é organizado como um pipeline experimental baseado em arquivos. Um plano YAML define o espaço experimental, incluindo modo de carga, rótulo do caminho de execução, tamanho do modelo, número de repetições e parâmetros de *runtime*. O *runner* expande esse plano em casos concretos, executa cada caso como um subprocesso e registra um diretório autocontido por execução, contendo `run.json`, `summary.json`, `power_samples.csv`, logs e métricas específicas do modo. A energia da GPU é amostrada por uma *thread* auxiliar via NVML, com `nvidia-smi` como *fallback*; a energia de CPU/sistema é coletada via RAPL quando disponível, ou por um estimador de melhor esforço caso contrário [David et al. 2010]. A energia é integrada a partir da série temporal de potência por meio da regra do trapézio. O GC-Bench também valida a cobertura experimental ao reconstruir, a partir do plano YAML, os casos esperados e compará-los com as execuções realizadas; execuções que resultam em *out-of-memory* são tratadas como resultados válidos de cobertura.

A carga de trabalho utiliza um mesmo backbone *TinyLM* em PyTorch para ambos os rótulos de execução. Na suíte fixa principal, o espaço efetivo de modelos concentra-se nas configurações Small e Large. O caminho `pytorch` executa o modelo diretamente em PyTorch [Paszke et al. 2019]. Já o caminho `go_rust` preserva a mesma computação em PyTorch, mas adiciona um estágio externo de tokenização, no qual um orquestrador em Go invoca repetidamente um microkernel em Rust sobre um corpus sintético. O treino utiliza lotes sintéticos, perda de entropia cruzada para predição do próximo token, AdamW e precisão mista (FP16/BF16). A inferência executa apenas o passo direto (*forward-only*) e registra latências p50/p95/p99 e energia de cauda. O modo H2D isola transferências de CPU para GPU sob diferentes configurações de memória *pinned* e transferências assíncronas.

A campanha principal utiliza a suíte `paper_ready_extended_fixed`, com 1.018 execuções validadas distribuídas entre treino, inferência, H2D, ablação e experimentos de sensibilidade. Uma suíte complementar, `paper_powercap_elevated`, adiciona 144 execuções com RAPL ativo e controle de limite de potência. Os experimentos foram conduzidos em uma NVIDIA GeForce RTX 5090 com driver 590.48.01 e driver CUDA 13010, combinada com um Intel Core Ultra 9 285K, Linux 6.17 e Python 3.12.3. As principais métricas são J/token da GPU, tokens/s/W, latência p50/p95/p99, energia de cauda, GB/s e J/GB. Os intervalos de confiança são estimados por *bootstrap*, e as comparações entre caminhos de execução utilizam o teste de Mann–Whitney U.

3. Resultados

A Tabela 2 resume os principais resultados agregados. No regime central de treino, os dois caminhos de execução apresentaram *throughput* muito próximo, mas o caminho `go_rust` foi ligeiramente mais eficiente nas métricas restritas à GPU: 0,002730 versus 0,002776 J/token, e 400,84 versus 393,76 tokens/s/W. Essa diferença, porém, é estatisticamente fraca: em `train_core_e2e_fixed`, nenhuma das oito comparações entre caminhos de execução atingiu $p < 0{,}05$ para J/token ou tokens/s/W, e apenas 1 de 16 comparações foi significativa na varredura entre regime computacional e ponta a ponta. A eficiência energética também depende fortemente do regime. Dobrar `batch_tokens` de 4096 para 8192 reduz substancialmente J/token em ambos os caminhos, enquanto lotes

Table 1. Espaço experimental principal.

Fator	Valores
Caminho de execução	`pytorch`, `go_rust`
Tamanho do modelo	Small, Large
Modo	train, infer, h2d
`seq_len`	256, 512
`batch_tokens`	4096, 8192
Execuções (suíte fixa)	1.018 validadas
Execuções (power-cap)	144 validadas

Table 2. Principais resultados agregados.

Modo	Caminho	J/token	tok/s/W	p99 (ms)	GB/s ou J/GB
Treino	`go_rust`	0.002730	400.84	–	–
Treino	`pytorch`	0.002776	393.76	–	–
Inferência	`go_rust`	0.002885	470.63	3.088	–
Inferência	`pytorch`	0.002703	499.27	3.086	–
H2D pageable	–	–	–	–	13.97 / 45.42
H2D pinned	–	–	–	–	40.77 / 37.66

menores e sequências mais longas tendem a piorar a eficiência. Esse efeito de regime é mais forte do que o próprio rótulo do caminho de execução. O compromisso entre velocidade e eficiência também não é monotônico: no estudo de ablação, alguns controles tornaram o sistema mais rápido, porém menos eficiente, enquanto outros melhoraram ambas as métricas para modelos menores.

Na inferência, a separação é mais clara. Em `infer_slo_fixed`, `pytorch` reduziu o J/token da GPU em 6,73% em relação a `go_rust`, melhorou tokens/s/W em 5,74% e elevou o *throughput* em 5,84%, enquanto a latência p99 permaneceu praticamente idêntica. Ambos os caminhos atenderam a todos os SLOs avaliados, e a análise de significância reforçou esse quadro para J/token e tokens/s/W.

Os microbenchmarks de H2D são dominados pela configuração de transferência, e não pelo rótulo do caminho de execução. O uso de memória *pinned* eleva a largura de banda de cerca de 14 GB/s para 37–41 GB/s e reduz a energia de aproximadamente 45–49 J/GB para 38 J/GB, resultando em um ganho médio de 2,82× e em uma redução de 19% em J/GB. Para *buffers* de pelo menos 64 MB, o ganho sobe para cerca de 3,06× e J/GB cai aproximadamente 32%. A energia de CPU também se mostra relevante quando medida de forma robusta. Na suíte `paper_powercap_elevated`, o RAPL esteve ativo em 144/144 execuções, e a CPU respondeu, em média, por 27,78% da energia total do sistema no treino e por 31,41% na inferência. A varredura de limite de potência mostra ainda que o ponto operacional mais eficiente não é fixo: esse ponto ótimo varia com o modo, o modelo e o caminho de execução.

4. Ameaças à validade

Este estudo apresenta três limitações principais. Primeiro, 970 das 1.018 execuções da suíte fixa principal acionaram o mecanismo de *warm-up* de contingência, de modo que algumas execuções curtas ainda podem conter efeitos de partida a frio, como inicialização de CUDA e transientes iniciais de *runtime*. Segundo, na suíte fixa, a energia de CPU colapsou efetivamente para zero; assim, as métricas de "sistema" reportadas nesse conjunto devem ser interpretadas, na prática, como métricas restritas à GPU. Portanto, conclusões que incluem CPU dependem da suíte separada de controle de potência, na qual o RAPL funcionou em 144/144 execuções. Terceiro, o caminho `go_rust` atual não corresponde a um backend nativo completo em Go/Rust para o modelo, mas a um caminho de execução em PyTorch acrescido de um estágio auxiliar de tokenização. De modo semelhante, algumas *flags* de ablação aceitas pela carga de trabalho não ativam *kernels* totalmente distintos em nível de produção; por isso, esses resultados devem ser interpretados como variações controladas da carga, e não como comparações estritas entre implementações independentes.

5. Conclusão

Apresentamos o GC-Bench, um *harness* reproduzível para avaliação energética de cargas de IA em GPU, combinando orquestração orientada por YAML, medição via NVML/RAPL e geração automatizada de relatórios estatísticos. Ao longo de 1.018 execuções validadas, o treino mostrou quase empate entre os dois caminhos de execução, enquanto, na inferência, o PyTorch puro foi claramente mais eficiente, reduzindo o J/token da GPU em 6,73% em relação ao caminho com tokenização externa em Go/Rust. O uso de memória *pinned* também se mostrou a otimização dominante para transferências H2D. Como trabalho futuro, pretende-se implementar um backend nativo em Go/Rust, estabilizar a coleta de energia de CPU e ampliar a reprodutibilidade por contêineres.

References

David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., and Le, C. (2010). Rapl: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, pages 189–194.

Mattson, P., Cheng, C., Diamos, G., Coleman, C., Micikevicius, P., Patterson, D., Tang, H., Wei, G., Bailis, P., Bittorf, V., Brooks, D., Chen, D., Dutta, D., Gupta, U., Hazelwood, K., Hock, A., Huang, X., Kang, D., Kanter, D., Kumar, N., Liao, J., Narayanan, D., Oguntebi, T., Pekhimenko, G., Pentecost, L., Reddi, V. J., Robie, T., John, T. S., Wu, C., Xu, L., Young, C., and Zaharia, M. (2020). Mlperf training benchmark. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 2.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32.

Patterson, D., Gonzalez, J., Le, Q. V., Liang, C., Munguia, L., Rothchild, D., So, D. R., Texier, M., and Dean, J. (2021). Carbon emissions and large neural network training. *CoRR*, abs/2104.10350.