

Avaliação de Desempenho e Uso de VRAM em GPUs: Um Estudo Comparativo entre JAX (XLA) e PyTorch no Vision Transformer

Martony Demes da Silva

Curso Superior de Tecnologia de Ciência de Dados
Universidade Federal do Mato Grosso do Sul (UFMS)

`martony.demes@ufms.br`

Abstract. *The scaling of Transformer architectures imposes hardware resource constraints on GPUs. This paper presents a comparative performance study between JAX (XLA) and PyTorch (Eager and Compiled) using a Multi-Head Self-Attention block from a Vision Transformer on an NVIDIA Tesla T4 GPU. Our empirical evaluations demonstrate that JAX achieves up to 3984.74 samples/s throughput with a notable reduction in peak VRAM consumption compared to PyTorch Eager, highlighting its resource efficiency in memory-constrained environments, despite the initial JIT compilation time.*

Resumo. *O dimensionamento de arquiteturas Transformer impõe restrições de recursos computacionais e de memória em GPUs. Este artigo apresenta um estudo comparativo de desempenho entre JAX (XLA) e PyTorch (Eager e Compiled) aplicando um bloco de Atenção Multicabeça de um Vision Transformer em uma GPU NVIDIA Tesla T4. Os resultados empíricos demonstram que o JAX atinge até 3984,74 amostras/s com uma redução observável na alocação de pico de VRAM em relação ao PyTorch Eager, evidenciando sua eficiência em cenários de memória limitada, a despeito do tempo inicial de compilação JIT.*

1. Introdução

O avanço das arquiteturas baseadas em *Transformers*, especialmente no domínio da Visão Computacional por meio dos *Vision Transformers* (ViT), estabeleceu novos patamares de precisão em aprendizado profundo. Contudo, a demanda computacional atrelada ao mecanismo de Atenção Multicabeça (*Multi-Head Self-Attention* – MHA) introduz gargalos operacionais de latência e consumo de memória VRAM em unidades de processamento gráfico [Dosovitskiy et al. 2021]. Diante disso, a seleção do arcabouço de desenvolvimento torna-se um fator determinante para a eficiência de execução.

Enquanto o PyTorch se consolidou pela facilidade de desenvolvimento no modo imperativo (*Eager*), bibliotecas Python de programação funcional como o JAX, alimentadas pelo compilador *Accelerated Linear Algebra* (XLA), aplicam otimizações de fusão de *kernels* (*kernel fusion*) via compilação *Just-In-Time* (JIT) [Bradbury et al. 2018]. Avaliações empíricas padronizadas são essenciais na Computação de Alto Desempenho para mensurar o impacto prático dessas abordagens [SBC 2021, SBC 2020]. Este trabalho analisa o *trade-off* de desempenho, vazão (*throughput*) e alocação de VRAM entre JAX e PyTorch sob variadas cargas de processamento.

2. Metodologia e Configuração Experimental

Para isolar o comportamento da GPU e mitigar potenciais gargalos de entrada e saída (I/O), implementou-se um bloco funcional isolado de MHA (dim = 768, heads = 12, seq_len = 196) correspondente à especificação do ViT-Base. A avaliação contemplou três configurações: **JAX** (com decorador `@jit`), **PyTorch Eager** e **PyTorch Compiled** (`torch.compile` empregando o backend Inductor).

O experimento foi conduzido em uma GPU NVIDIA Tesla T4 (16 GB GDDR6, arquitetura Turing), com driver versão 580.82.07, CUDA 12.8, PyTorch 2.11 e JAX 0.7.2. Para assegurar a reprodutibilidade dos experimentos, o código-fonte e o ambiente de execução foram disponibilizados publicamente¹. O protocolo experimental adotou sementes estáticas (42) e as seguintes diretrizes de medição:

1. **Fase de Aquecimento (*Warm-up*):** As primeiras 20 iterações foram desconsideradas dos cálculos estatísticos para isolar o custo da compilação JIT do estado estacionário.
2. **Sincronização de Execução Assíncrona:** Empregou-se o método `.block_until_ready()` no JAX e a função `torch.cuda.synchronize()` no PyTorch ao término de cada passo para evitar medições antecipadas no lado do hospedeiro (CPU).
3. **Variação de Carga:** Avaliaram-se tamanhos de lote (*Batch Sizes*) $B \in \{32, 64, 128\}$ durante 100 iterações contínuas em estado estacionário.

3. Resultados e Discussão

A Tabela 1 apresenta os dados consolidados de latência, vazão, tempo de primeira iteração (*First Step*) e pico de memória alocada. O comportamento comparativo entre as abordagens está ilustrado na Figura 1.

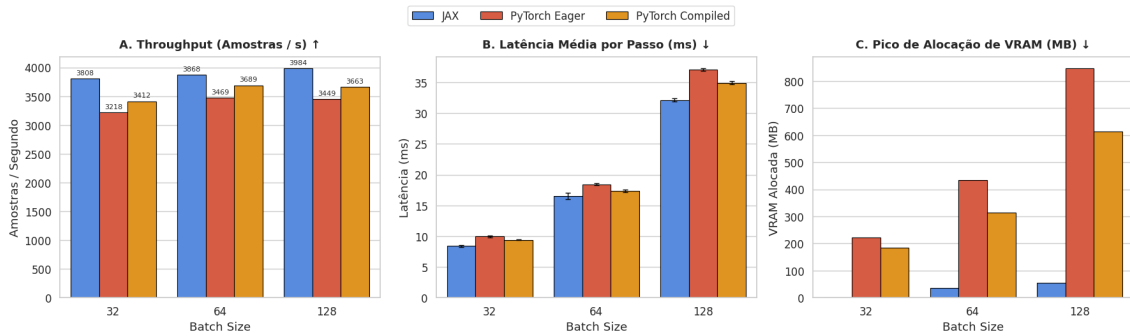


Figura 1. Desempenho comparativo entre JAX, PyTorch Eager e PyTorch Compiled sob variação do *Batch Size*: (A) Vazão em amostras/s; (B) Latência média em ms; (C) Pico de alocação de VRAM em MB.

Em relação à latência de execução e à vazão (Figuras 1A e 1B), o JAX obteve menores tempos médios por iteração em todas as cargas avaliadas. Para $B = 128$, o JAX registrou latência de 32,12 ms, em comparação a 37,10 ms no PyTorch Eager e 34,94

¹https://colab.research.google.com/drive/1RI8H_JL3_6UGTuAypbVw_wMI5FBjC1Zb#scrollTo=NyMeAOETmGIv

Tabela 1. Resultados do Benchmark em GPU Tesla T4 sob variação do Batch Size.

Batch Size (B)	Framework	Latência (ms)	Desvio Padrão (ms)	Vazão (amostras/s)	1º Passo (ms)	Pico VRAM (MB)
32	JAX	8.402	0.125	3808.44	914.06	0.00*
32	PyTorch Eager	9.942	0.194	3218.52	125.35	222.42
32	PyTorch Compiled	9.377	0.057	3412.46	9389.75	184.82
64	JAX	16.543	0.530	3868.71	947.96	35.50
64	PyTorch Eager	18.444	0.164	3469.99	14.73	435.59
64	PyTorch Compiled	17.347	0.170	3689.40	1843.72	314.05
128	JAX	32.123	0.247	3984.74	1678.54	54.50
128	PyTorch Eager	37.103	0.188	3449.87	29.97	847.25
128	PyTorch Compiled	34.943	0.254	3663.08	27.88**	612.62

*O valor nulo decorre da pré-alocação estática do buffer do XLA durante o contexto inicial.

**O tempo reduzido do 1º Passo no PyTorch Compiled em $B = 128$ decorre da reutilização do grafo pré-compilado em $B = 32$.

ms no PyTorch Compiled. Esse ganho de eficiência traduziu-se em uma vazão superior, atingindo 3984,74 amostras/s no JAX frente a 3449,87 no modo Eager do PyTorch.

A análise do consumo de memória VRAM (Figura 1C) indica diferenças na gestão de tensores intermediários. Enquanto a alocação no PyTorch Eager escalou até 847,25 MB em $B = 128$, o JAX apresentou alocação incremental de 54,50 MB. Esse perfil é atribuído à otimização do compilador XLA, que realiza análise estática do grafo de computação para reutilizar regiões de memória de forma contínua durante a execução.

Por outro lado, verifica-se o comportamento do custo de compilação no primeiro passo (*First Step*). Na execução inicial ($B = 32$), o PyTorch Compiled requereu 9389,75 ms devido à compilação completa do grafo de computação e geração de *kernels* C++/Triton via *backend* Inductor. Nas variações subsequentes ($B = 64$ e $B = 128$), o tempo do primeiro passo no PyTorch Compiled reduziu-se drasticamente para 1843,72 ms e 27,88 ms, respectivamente. Essa redução expressiva decorre da reutilização e adaptação do grafo previamente otimizado na memória do processo, evitando o *re-jit* integral. Em contrapartida, o JAX apresentou um tempo de compilação JIT inicial de 914,06 ms ($B = 32$), escalando para 1678,54 ms em $B = 128$ devido ao custo de alocação de buffers para maiores tamanhos de lote. O modo Eager do PyTorch manteve menor latência no primeiro passo em todas as cargas (entre 14,73 ms e 125,35 ms) por prescindir da etapa de síntese prévia de grafo.

4. Conclusão

Os resultados obtidos indicam que a utilização do JAX com compilação XLA em blocos de Atenção Multicabeça proporciona ganhos de vazão e menor ocupação de memória VRAM na GPU Tesla T4 em comparação às variantes do PyTorch. Tais características tornam o JAX uma alternativa viável para cenários de inferência contínua ou ambientes com restrição de memória física. No entanto, o custo computacional da compilação JIT inicial (JAX) e do aquecimento do *backend* Inductor (PyTorch Compiled) deve ser considerado na escolha do ecossistema: enquanto o PyTorch Compiled amortiza o tempo de compilação após as primeiras execuções, o PyTorch Eager permanece ideal para prototipagem rápida e depuração interativa. Como trabalhos futuros, pretende-se expandir a análise para topologias completas de modelos e arquiteturas multi-GPU.

Referências

- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., et al. (2018). JAX: composable transformations of Python+NumPy programs. *<http://github.com/google/jax>*, 1(1):0–1.
- Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., et al. (2021). An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations (ICLR)*.
- SBC, editor (2020). *Anais da VI Escola Regional de Alto Desempenho do Centro-Oeste (ERAD-CO 2020)*. Sociedade Brasileira de Computação (SBC).
- SBC, editor (2021). *Anais da VII Escola Regional de Alto Desempenho do Centro-Oeste (ERAD-CO 2021)*. Sociedade Brasileira de Computação (SBC).