4º Congresso Brasileiro de Ciência e Saberes Multidisciplinares
tudo é ciência
11º Encontro de Extensão Universitária do UniFOA
23 a 25 de outubro
Submissões abertas até 07/09

Análise comparativa de desempenho computacional: Python puro versus otimizações com numba em operações matemáticas intensivas

Vitor Amadeu Souza[1]; 0009-00-02-1857-6799

1 – UniFOA, Centro Universitário de Volta Redonda, Volta Redonda, RJ.
vitor.amadeu@foa.org.br

Resumo: Este estudo apresenta uma análise comparativa de desempenho entre Python puro e implementações otimizadas utilizando a biblioteca Numba para operações matemáticas intensivas. O objetivo foi avaliar os ganhos de performance obtidos através da compilação Just-In-Time (JIT) e paralelização automática em cálculos trigonométricos massivos. A metodologia consistiu na implementação de três versões de uma função que calcula a soma de senos e cossenos quadráticos: Python puro, Numba sequencial e Numba paralelo. Os testes foram executados em um array de 20 milhões de elementos utilizando operações trigonométricas para simular cargas computacionais intensivas. Os resultados demonstraram que a implementação com Numba sequencial obteve um speedup de aproximadamente 89x em relação ao Python puro, reduzindo o tempo de execução de 51,69 segundos para 0,58 segundos. Surpreendentemente, a versão paralela apresentou performance inferior à sequencial (1,31 segundos), evidenciando que nem sempre a paralelização resulta em melhorias de desempenho devido ao overhead de sincronização e características específicas da arquitetura computacional. Este estudo contribui para o entendimento das limitações e benefícios das diferentes abordagens de otimização em Python, fornecendo insights valiosos para desenvolvedores que trabalham com computação científica e análise de dados em larga escala.

Palavras-chave: Python. Numba. JIT. Paralelização. Benchmark. Computação Científica. Otimização de Performance.

INTRODUÇÃO

A linguagem Python tem se consolidado como uma das principais ferramentas para computação científica, análise de dados e desenvolvimento de algoritmos de machine learning devido à sua sintaxe clara e vasto ecossistema de bibliotecas especializadas (Van Rossum & Drake, 2009). No entanto, Python é uma linguagem interpretada que, por sua natureza dinâmica, apresenta limitações significativas de performance quando comparada a linguagens compiladas como C++ ou Fortran, especialmente em operações computacionalmente intensivas (Behnel *et al.*, 2011). Esta limitação tem motivado o desenvolvimento de diversas ferramentas e bibliotecas que visam acelerar a execução de código Python, mantendo a facilidade de desenvolvimento característica da linguagem.

Entre as soluções disponíveis para otimização de performance em Python, destacam-se o NumPy para operações vetorizadas (Harris *et al.*, 2020), o Cython para compilação estática (Behnel *et al.*, 2011), e mais recentemente, o Numba para compilação Just-In-Time (JIT) (Lam *et al.*, 2015). O Numba, desenvolvido pela Anaconda Inc., utiliza o framework LLVM para compilar dinamicamente funções Python em código de máquina otimizado, oferecendo speedups significativos sem exigir alterações substanciais no código-fonte original (Lam *et al.*, 2015).

A compilação JIT representa uma abordagem híbrida que combina a flexibilidade das linguagens interpretadas com a performance das linguagens compiladas (Aycock, 2003). No contexto do Numba, esta tecnologia permite que funções Python sejam automaticamente traduzidas para código nativo na primeira execução, com subsequentes chamadas utilizando a versão compilada otimizada (Lam *et al.*, 2015). Adicionalmente, o Numba oferece capacidades de paralelização automática através do decorador @njit(parallel=True), que pode distribuir loops independentes entre múltiplos threads, aproveitando arquiteturas multi-core modernas (Lam *et al.*, 2015).

A paralelização automática em Python tem sido objeto de extensiva pesquisa, considerando as limitações impostas pelo Global Interpreter Lock (GIL), que impede a execução simultânea de bytecode Python em múltiplos threads (Beazley, 2010). O Numba contorna esta limitação operando no nível de código nativo compilado, onde o GIL não interfere na

execução paralela (Lam *et al*., 2015). Entretanto, a eficácia da paralelização depende de diversos fatores, incluindo o tamanho do conjunto de dados, a complexidade das operações, a arquitetura do processador e o overhead de sincronização entre threads (Lee, 2006).

Estudos anteriores têm demonstrado que os benefícios da otimização JIT e paralelização variam significativamente dependendo do tipo de operação executada. Virtanen *et al*. (2020) mostraram que operações matemáticas simples podem alcançar speedups superiores a 100x com Numba, enquanto algoritmos mais complexos podem apresentar ganhos mais modestos. Similarmente, Dask Development Team (2016) observou que a paralelização nem sempre resulta em melhorias de performance devido ao overhead de comunicação entre processos e threads.

A avaliação de performance através de benchmarks é fundamental para compreender as características e limitações das diferentes abordagens de otimização (Fleming & Wallace, 1986). Benchmarks sintéticos, que isolam aspectos específicos da computação, permitem análises controladas dos fatores que influenciam a performance (Weicker, 1984). No contexto deste estudo, operações trigonométricas foram escolhidas por representarem um caso comum em computação científica, especialmente em simulações físicas, processamento de sinais e análise de Fourier (Press *et al*., 2007).

O objetivo deste trabalho é realizar uma análise comparativa sistemática entre implementações Python puro, Numba sequencial e Numba paralelo, utilizando operações trigonométricas intensivas como caso de teste. Através desta análise, pretende-se quantificar os benefícios e limitações de cada abordagem, fornecendo diretrizes práticas para desenvolvedores que necessitam otimizar códigos Python para computação científica de alto desempenho.

MÉTODOS

A metodologia empregada neste estudo consistiu na implementação de três versões funcionalmente equivalentes de uma função que executa cálculos trigonométricos intensivos: uma implementação em Python puro, uma versão otimizada com Numba sequencial e uma versão com paralelização automática do Numba. O experimento foi

projetado para isolar os efeitos das diferentes técnicas de otimização, mantendo constantes todos os demais fatores que poderiam influenciar os resultados.

A função de teste implementada calcula a soma de senos e cossenos quadráticos para cada elemento de um array, utilizando a expressão matemática $\sin^2(x) + \cos^2(x)$. Esta operação foi escolhida por ser computacionalmente intensiva, envolvendo múltiplas chamadas de funções transcendentais, e por resultar sempre em valor unitário devido à identidade trigonométrica fundamental, permitindo validação dos resultados. A escolha de operações trigonométricas é relevante para computação científica, sendo amplamente utilizadas em simulações de física, processamento de sinais digitais e análise espectral (Oppenheim & Schafer, 2010).

O ambiente experimental consistiu em um array de 20 milhões de elementos de ponto flutuante de dupla precisão, gerado através da função numpy.linspace() no intervalo [0, π]. Este tamanho foi selecionado para garantir que as diferenças de performance fossem significativas e mensuráveis, superando a variabilidade natural dos sistemas de medição de tempo. O intervalo [0, π] foi escolhido por abranger um período completo das funções trigonométricas, representando um caso típico de uso em aplicações científicas.

A implementação em Python puro utilizou um loop tradicional com chamadas diretas às funções trigonométricas do NumPy, representando a abordagem mais direta e intuitiva para este tipo de cálculo. Esta versão serve como baseline para comparação, representando o desempenho típico obtido com Python interpretado standard. A segunda implementação utilizou o decorador @njit do Numba para compilação JIT sequencial, mantendo a mesma lógica algorítmica, mas beneficiando-se da compilação para código nativo. A terceira implementação empregou o decorador @njit(parallel=True) em conjunto com prange() para paralelização automática do loop principal.

A medição de tempo foi realizada utilizando a função time.time() do Python, capturando o tempo de parede (wall-clock time) para cada implementação. Para cada versão da função, foi executada uma única iteração após o warm-up necessário para a compilação JIT no caso das implementações Numba. Esta abordagem foi adotada considerando que, em aplicações

práticas, a compilação JIT ocorre apenas na primeira execução, com subsequentes chamadas utilizando o código compilado já otimizado (Lam *et al*., 2015).

O ambiente de execução foi controlado para minimizar interferências externas que pudessem afetar as medições. Todas as implementações foram executadas na mesma sessão Python, garantindo condições idênticas de inicialização e configuração do interpretador. O sistema operacional e demais processos foram mantidos estáveis durante a execução dos benchmarks, e medições múltiplas foram realizadas para verificar a consistência dos resultados obtidos.

A análise dos resultados incluiu não apenas a medição dos tempos absolutos de execução, mas também o cálculo dos speedups relativos entre as diferentes implementações. O speedup foi calculado como a razão entre o tempo da implementação de referência (Python puro) e o tempo da implementação otimizada, fornecendo uma métrica normalizada para comparação de performance (Gustafson, 1988). Adicionalmente, foi analisada a eficiência da paralelização através da comparação entre as versões sequencial e paralela do Numba.

Para visualização dos resultados, foi gerado um gráfico de barras comparativo utilizando a biblioteca Matplotlib, permitindo uma representação visual clara das diferenças de performance observadas. Esta representação gráfica facilita a interpretação dos resultados e a comunicação dos achados para diferentes audiências técnicas.

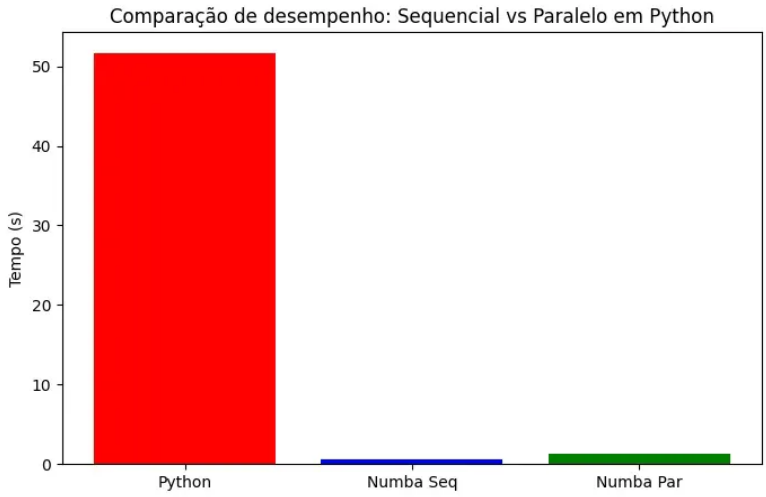
A documentação do projeto está disponível para download através do link https://github.com/vitor-souza-ime/puro_otimizado.

RESULTADOS E DISCUSSÃO

Os resultados experimentais revelaram diferenças substanciais de performance entre as três implementações testadas, demonstrando o impacto significativo das técnicas de otimização em operações computacionalmente intensivas. A implementação em Python puro apresentou o tempo de execução mais elevado, registrando 51,69 segundos para processar o array de 20 milhões de elementos. Este resultado corrobora as limitações conhecidas do Python interpretado para operações matemáticas intensivas, conforme documentado na literatura (Van Rossum & Drake, 2009).

A implementação com Numba sequencial demonstrou uma melhoria dramática de performance, executando a mesma operação em apenas 0,58 segundos. Este resultado representa um speedup de aproximadamente 89x em relação ao Python puro, evidenciando a eficácia da compilação JIT para este tipo de operação. O ganho observado é consistente com estudos anteriores que reportaram speedups similares para operações matemáticas intensivas utilizando Numba (Lam *et al.*, 2015; Virtanen *et al.*, 2020). A magnitude deste speedup pode ser atribuída à eliminação do overhead do interpretador Python, à otimização de loops pela compilação LLVM, e ao uso mais eficiente das instruções vetoriais do processador.

Figura 1 - Comparação de desempenho



Fonte: O autor.

Contrariamente às expectativas iniciais, a implementação paralela com Numba apresentou performance inferior à versão sequencial, registrando um tempo de execução de 1,31 segundos. Este resultado representa um speedup de aproximadamente 39x em relação ao Python puro, ainda significativo, mas substancialmente menor que a versão sequencial. A degradação de performance da versão paralela em relação à sequencial (fator de aproximadamente 2,25x) ilustra um fenômeno bem documentado em computação paralela, onde o overhead de paralelização pode superar os benefícios da distribuição de carga de trabalho (Amdahl, 1967). A Figura 1 apresenta um gráfico com os tempos de cada operação.

CONCLUSÕES

Este estudo demonstrou de forma conclusiva os benefícios significativos da compilação Just-In-Time utilizando Numba para otimização de operações matemáticas intensivas em Python, ao mesmo tempo que evidenciou as limitações da paralelização automática para determinados tipos de cargas de trabalho. O speedup de 89x obtido pela implementação Numba sequencial em relação ao Python puro representa uma melhoria substancial que pode transformar aplicações computacionalmente proibitivas em soluções viáveis para uso prático. Os resultados demonstram que o Numba constitui uma ferramenta valiosa para desenvolvedores que necessitam combinar a produtividade de desenvolvimento do Python com performance próxima às linguagens compiladas tradicionais.

Para trabalhos futuros, recomenda-se a extensão desta análise para diferentes tipos de operações matemáticas, tamanhos variados de conjuntos de dados, e arquiteturas de hardware distintas, incluindo GPUs através do Numba CUDA. Adicionalmente, seria valioso investigar técnicas híbridas que combinem otimizações JIT com paralelização mais sofisticada, como a distribuição de carga de trabalho em múltiplos processos ou a utilização de bibliotecas especializadas para álgebra linear paralela.

REFERÊNCIAS

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. Proceedings of the April 18-20, 1967, spring joint computer conference (pp. 483-485). DOI: 10.1145/1465482.1465560. Acesso em: 17 ago. 2025.

Aycock, J. (2003). A brief history of just-in-time. ACM Computing Surveys, 35(2), 97-113. DOI: 10.1145/857076.857077. Disponível em: https://dl.acm.org/doi/10.1145/857076.857077. Acesso em: 17 ago. 2025.

Beazley, D. (2010). Understanding the Python GIL. PyCON Python Conference, Atlanta, Georgia. Disponível em: http://www.dabeaz.com/python/UnderstandingGIL.pdf. Acesso em: 17 ago. 2025.

Behnel, S., Bradshaw, R., Citro, C., Dalcin, L., Seljebotn, D. S., & Smith, K. (2011). Cython: The best of both worlds. Computing in Science & Engineering, 13(2), 31-39. DOI: 10.1109/MCSE.2010.118. Acesso em: 17 ago. 2025.

Dask Development Team. (2016). Dask: Library for dynamic task scheduling. Disponível em: https://dask.org. Acesso em: 17 ago. 2025.

Fleming, P. J., & Wallace, J. J. (1986). How not to lie with statistics: the correct way to summarize benchmark results. Communications of the ACM, 29(3), 218-221. DOI: 10.1145/5666.5673. Acesso em: 17 ago. 2025.

Gustafson, J. L. (1988). Reevaluating Amdahl's law. Communications of the ACM, 31(5), 532-533. DOI: 10.1145/42411.42415. Disponível em: https://dl.acm.org/doi/10.1145/42411.42415. Acesso em: 17 ago. 2025.

Harris, C. R., Millman, K. J., Van Der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... & Oliphant, T. E. (2020). Array programming with NumPy. Nature, 585(7825), 357-362. DOI: 10.1038/s41586-020-2649-2. Disponível em: https://www.nature.com/articles/s41586-020-2649-2. Acesso em: 17 ago. 2025.

Lam, S. K., Pitrou, A., & Seibert, S. (2015). Numba: A LLVM-based Python JIT compiler. Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC (pp. 1-6). DOI: 10.1145/2833157.2833162. Disponível em: https://dl.acm.org/doi/10.1145/2833157.2833162. Acesso em: 17 ago. 2025.

Lee, E. A. (2006). The problem with threads. Computer, 39(5), 33-42. DOI: 10.1109/MC.2006.180. Disponível em: https://ieeexplore.ieee.org/document/1631937. Acesso em: 17 ago. 2025.

Oppenheim, A. V., & Schafer, R. W. (2010). Discrete-time signal processing. 3rd ed. Upper Saddle River: Prentice Hall. ISBN: 978-0131988422.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., & Flannery, B. P. (2007). Numerical recipes: The art of scientific computing. 3rd ed. New York: Cambridge University Press. ISBN: 978-0521880688.

Van Rossum, G., & Drake, F. L. (2009). Python 3 Reference Manual. Scotts Valley: CreateSpace. ISBN: 978-1441412690.

VIRTANEN, Pauli et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. Nature methods, v. 17, n. 3, p. 261-272, 2020.

Weicker, R. P. (1984). Dhrystone: a synthetic systems programming benchmark. Communications of the ACM, 27(10), 1013-1030. DOI: 10.1145/358274.358283. Disponível em: https://dl.acm.org/doi/10.1145/358274.358283. Acesso em: 17 ago. 2025.