



# CPython, Python e sua aplicação de alto desempenho

Acelerando computação científica com processamento paralelo

Python Brasil 2025

Esbel Tomas Valero Orellana

[evalero@uesc.br](mailto:evalero@uesc.br)

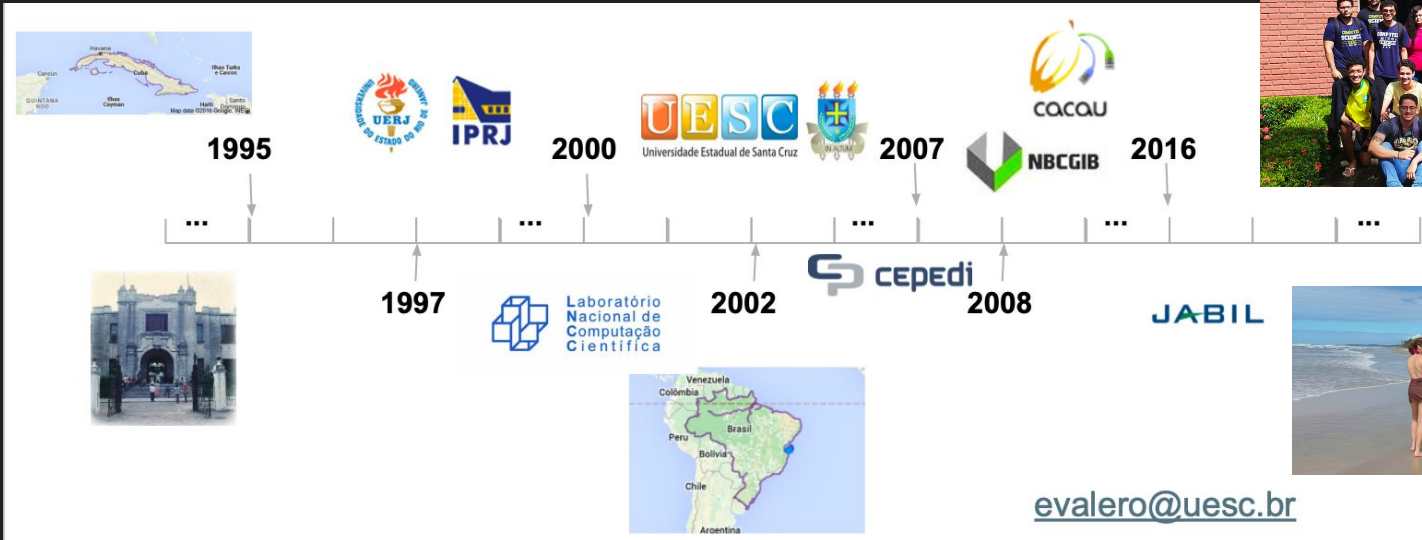


# O Palestrante

Nome: Esbel T. Valero Orellana

Mestrado e Doutorado em Modelagem Computacional

Professor Pleno DCET/UESC



[evalero@uesc.br](mailto:evalero@uesc.br)

# Agenda Acadêmica

- |    |   |    |  |
|----|---|----|--|
| 01 | O Desafio do Desempenho em Python         | 02 | CPython e Global Interpreter Lock (GIL)        |
| 03 | Multiplicação de Matrizes: Caso de Estudo | 04 | Processo de Multiplicação: BLAS & GEMM         |
| 05 | NumPy como Baseline de Performance        | 06 | OpenMP: Paralelização em Memória Compartilhada |
| 07 | CUDA: Computação em GPU                   | 08 | CPython: Ponte entre Python e C/C++            |
| 09 | Testes de Desempenho: Metodologia         | 10 | Análise e Discussão                            |



Pesquisa &  
Desenvolvimento

## Multiplicação de Matrizes: Caso de Estudo

- Uma matriz é um conjunto retangular de números, símbolos ou expressões, organizados em linhas e colunas.
- De forma geral dizemos que  $A_{m \times n}$  é uma matriz de m linhas e n colunas, onde m e n são inteiros positivos.
- Multiplicação de matrizes é uma operação binária que produz uma matriz a partir da multiplicação de outras duas.

$$C_{M \times N} = \alpha A_{M \times K} \times B_{K \times N} + \beta C_{M \times N}$$

$$C_{M \times N} = A_{M \times K} \times B_{K \times N}$$

$$C_{i,j} = \sum_{k=1}^K A_{i,k} B_{k,j}$$

# Multiplicação de Matrizes: Caso de Estudo



## Machine Learning

Fundamental em redes neurais, treinamento de modelos e operações de convolução.



## Análise de Dados

Processamento de grandes datasets, transformações lineares e estatísticas multivariadas.



## Modelagem Computacional

Simulações numéricas, sistemas lineares e métodos de elementos finitos.

**Python puro:** Loops aninhados são extremamente ineficientes • **NumPy:** Resolve com operações vetorizadas em C •  
**Bibliotecas customizadas:** Otimizações específicas com paralelização

# Processo de Multiplicação: BLAS & GEMM

## BLAS (Basic Linear Algebra Subprograms)

- Nível 1: Operações vetoriais (AXPY, DOT)
- Nível 2: Operações matriz-vetor (GEMV)
- Nível 3: Operações matriz-matriz (GEMM)

### GEMM - General Matrix Multiplication

$C = \alpha AB + \beta C$ , onde  $\alpha, \beta$  são escalares. Operação fundamental para Deep Learning.

## How to Optimize GEMM

### 1. Blocking

Divisão em blocos para cache L1/L2

### 2. Vectorization

Instruções SIMD (AVX, SSE) para paralelismo

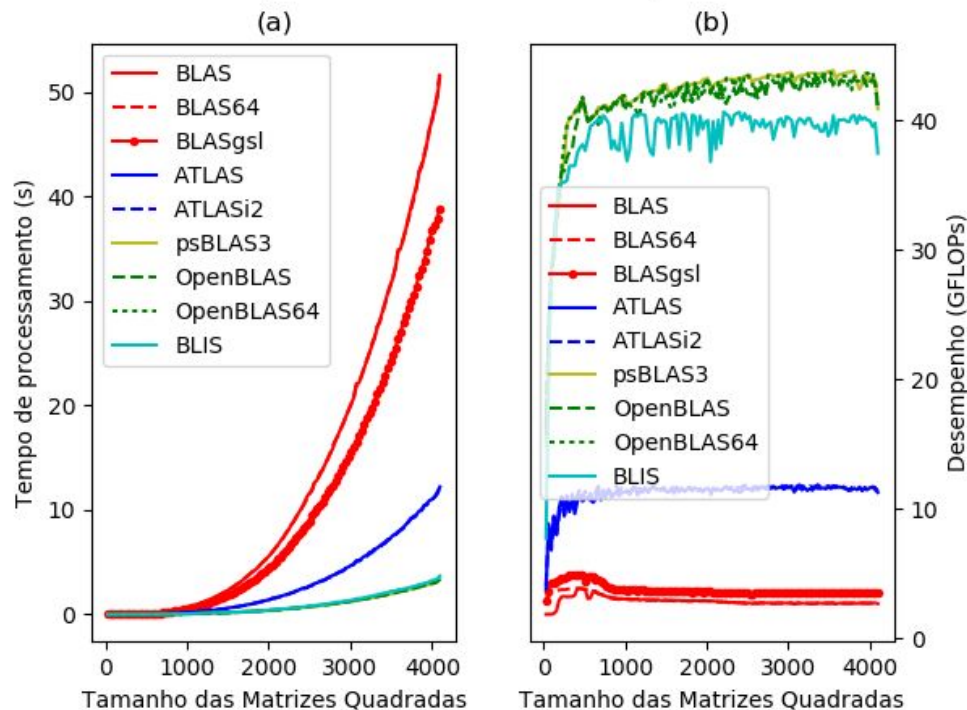
### 3. Loop Unrolling

Reduz overhead de controle de loops

**Referência:** How to Optimize GEMM • **Implementações:** OpenBLAS, Intel MKL, cuBLAS • **Aplicações:** NumPy, TensorFlow, PyTorch

# Multiplicação de Matrizes

Deempenho da Rotina DGEMM Sequenciais



# Multiplicação de Matrizes

```
1 def matmul_py(A, B):
2     """
3     Multiplicação de matrizes (listas de listas) 3 loops em Python puro.
4     A e B devem ser listas de listas de float (ou ints).
5     """
6     n = len(A)
7     m = len(B[0])
8     p = len(B)
9     p_ = len(A[0])
10    assert p == p_, "Dimensões inválidas para multiplicação de matrizes."
11    C = [[0.0]*m for _ in range(n)]
12    for i in range(n,0):
13        for j in range(m):
14            for k in range(p):
15                C[i][j] += A[i][k] * B[k][j]
16    return C
17
18 # Teste pequeno para validar a função
19
20 n = 64
21 A_small = [[random() for i in range(n)] for j in range(n)]
22 B_small = [[random() for i in range(n)] for j in range(n)]
23 C_small = matmul_py(A_small, B_small)
24 print(f"OK: C_small shape = {len(C_small)}x{len(C_small[0])}")
```



# O Desafio do Desempenho em Python



## Python Interpretado

Linguagem interpretada com limitações nativas para computação intensiva.



## Global Interpreter Lock (GIL)

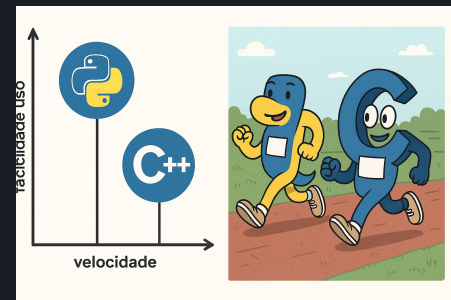
Impede paralelização verdadeira, limitando uso de múltiplos cores.



## Solução Híbrida

Combinar Python com C/C++ através de bibliotecas otimizadas.

Fonte: Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J. P., & Saraiva, J. (2017). Energy efficiency across programming languages: How do energy, time, and memory relate. SLE 2017 - Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, Co-Located with SPLASH 2017, 256-267. [DOI] (<https://doi.org/10.1145/3136014.3136031>)



## Desafio de Performance

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

# O que é o CPython?

## O que CPython?

- Implementação de referência da linguagem Python (intérprete + runtime).

### Escrito em C

É o programa que lê, interpreta e executa o código Python — por isso o nome C + Python  
compila .py → bytecode → executa na Python Virtual Machine.

## Impacto no Desempenho

Gerenciamento de memória: contagem de referências + coletor de lixo (ciclos).

C API expõe estruturas (PyObject\*) para criar/estender tipos e módulos nativos.

Base da distribuição oficial: CPython é o 'Python' que usamos no dia a dia.

# Global Interpreter Lock: O Gargalo do Python

## O que é o GIL?

- Mutex que protege objetos Python de acesso concorrente
- Permite apenas uma thread executar bytecode por vez
- Existe desde o Python 1.4 (1997)

### Por que o GIL existe?

Simplifica gerenciamento de memória, evita condições de corrida, melhora performance para operações single-thread.

## Impacto no Desempenho

### CPU-bound

Threads não aproveitam múltiplas cores para computação intensiva.

### I/O-bound

Threads liberam GIL durante I/O, permitindo alguma concorrência.

### Alternativas

Multiprocessing, Cython, NumPy, bibliotecas C/C++ com GIL liberado.

**Solução:** Bibliotecas C/C++ liberam o GIL durante computação intensiva • **Exemplos:** NumPy, OpenCV, TensorFlow • **Vantagem:** Paralelização real sem complexidade

# Multiplicação de Matrizes

```
1 def matmul_py_array(A: np.ndarray, B: np.ndarray) -> np.ndarray:
2     """
3     Multiplicação de matrizes usando loops Python puros e arrays NumPy.
4     A e B devem ser np.ndarray 2D.
5     """
6     m, n = A.shape
7     n, p = B.shape
8     assert n == p, "Dimensões inválidas para multiplicação de matrizes."
9     C = np.zeros((m, p))
10
11     for i in range(m):
12         for j in range(p):
13             for k in range(n):
14                 C[i, j] += A[i, k] * B[k, j]
15     return C
16
17 # Teste pequeno para validar a função
18 C_small_array = matmul_py_array(np.array(A_small), np.array(B_small))
19 print(f"OK: C_small_array shape = {C_small_array.shape}")
```

# NumPy como **Baseline** de Performance



## Implementação em C

Operações eficientes através de código compilado e vetorizado.



## BLAS Otimizado

Basic Linear Algebra Subprograms para operações matriciais.



## Padrão Ouro

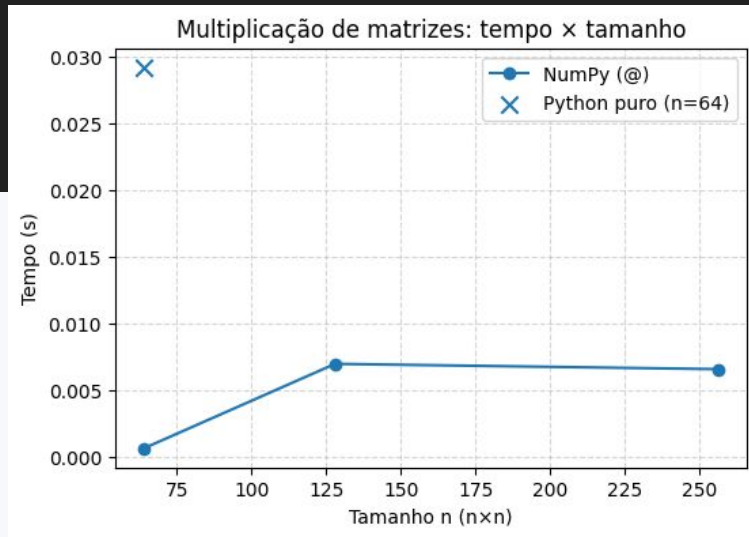
Operação @ ou np.dot() representa baseline para performance.

## Exemplo NumPy

```
import numpy as np # Criar matrizes A =  
np.random.rand(1000, 1000) B =  
np.random.rand(1000, 1000) # Multiplicação eficiente  
C = A @ B # ou np.dot(A, B)
```

# Multiplicação de Matrizes

```
1 def bench_numpy(sizes=SIZES, repeats=REPEATS, dtype=np.float64):
2     results = []
3     for n in sizes:
4         times = []
5         for _ in range(repeats):
6             A = np.random.rand(n, n).astype(dtype, copy=False)
7             B = np.random.rand(n, n).astype(dtype, copy=False)
8             t0 = time.time()
9             C = A @ B
10            times.append(time.time() - t0)
11            results.append({"n": n, "mean": float(np.mean(times)), "std": float(np.std(times)), "dtype": str(dtype)})
12    return results
13
14 numpy_results = bench_numpy()
15 numpy_results
```



# CPython: Ponte entre Python e C/C++



Python C API



Módulos Nativos



Wrappers C/C++

Interface para criar módulos de extensão em C/C++. Comportamento idêntico a módulos Python nativos. Exposição de funções C/C++ para uso em Python.

## Vantagens da Abordagem Híbrida

### Facilidade de Uso

Interface Python mantida, sem complexidade para o usuário.

### Performance Máxima

Código compilado com otimizações específicas do hardware.

# Extensões via API do CPython (C/C++)

## Como funciona?

Crie módulos nativos em C/C++ que se comportam como módulos Python.

## Passos:

definir funções (`PyCFunction`), tabela `PyMethodDef` e módulo (`PyModuleDef`).

## Como fazer?

Converter argumentos/retornos:

`PyArg_ParseTuple` / `Py_BuildValue`.

Liberar o GIL durante o cálculo pesado:

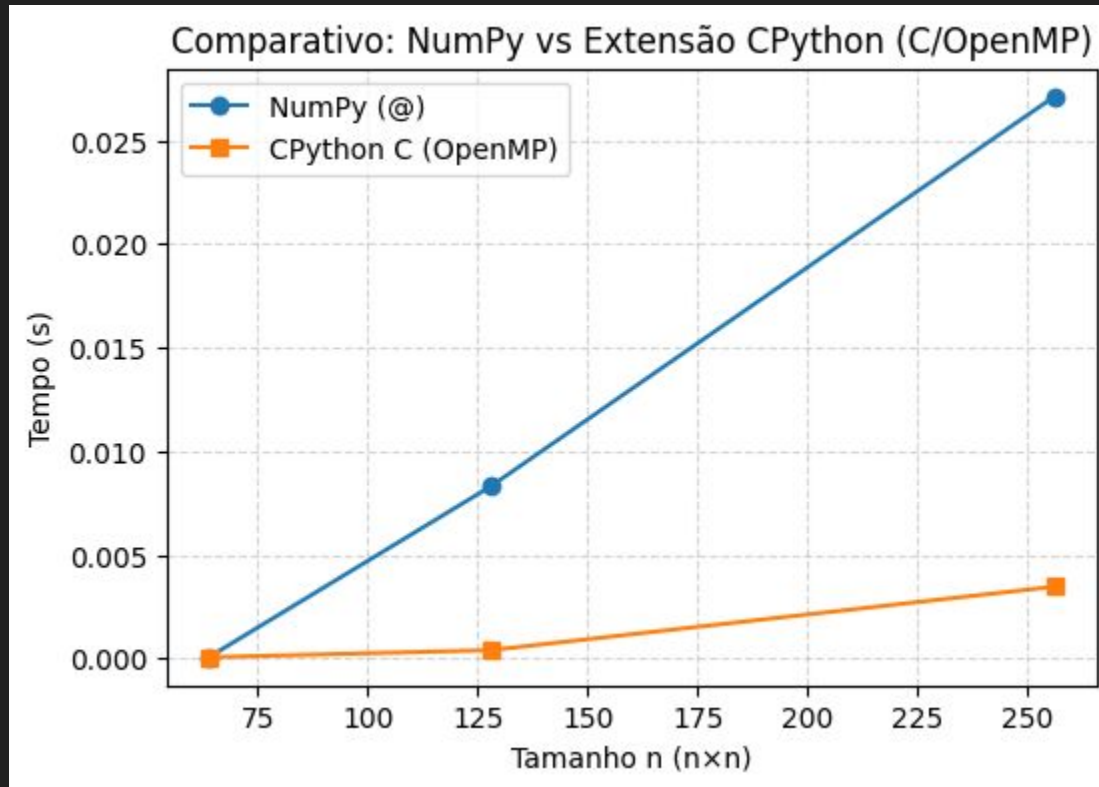
`Py_BEGIN_ALLOW_THREADS` / `Py_END_ALLOW_THREADS`.

Distribuição:

compilar com `setuptools`/`pyproject` e publicar como `wheel`.



## Extensões via API do CPython (C/C++)



# Integração via `ctypes` (FFI)

## Como funciona?

Carrega bibliotecas compartilhadas (.so/.dll) direto do Python (Foreign Function Interface).

Mapeia tipos  $C \longleftrightarrow$  Python: define `.argtypes` e `.restype` para segurança e performance.

Ideal para prototipagem rápida e ligação de funções C existentes (inclui chamadas a OpenMP/CUDA).

## Características

Menor acoplamento ao runtime do CPython (sem C API), mas menos controle sobre objetos Python.

Pode exigir conversões/cópias (`NumPy`  $\leftrightarrow$  C) se não usar ponteiros/contiguidade.

## Integração via `ctypes` (FFI)

- FFI (Foreign Function Interface): interface para chamar funções nativas C/C++ a partir do Python.
- Integração binária direta com bibliotecas compiladas (.so, .dll, .dylib), sem recompilar o interpretador.
- ``ctypes`` é o FFI padrão do Python: carrega libs, define tipos C  $\leftrightarrow$  Python (`argtypes/restype`) e faz conversões.
- Permite usar código de alto desempenho (OpenMP, BLAS, CUDA) com chamadas simples do Python.
- Excelente para protótipos e bindings leves; complementa a C API (mais poderosa, porém mais complexa).

# Comparando CPython C API × ctypes

Técnica	Nível de integração	Complexidade	Quando usar
ctypes	Carrega biblioteca existente	Baixa/Moderada	Já existe uma .so/.dll pronta, precisa somente chamar C
Extensão via C API	Integração profunda com Python	Alta	Precisa definir módulos , objetos Python

# Paralelização em Memória Compartilhada



OpenMP

## Diretivas de Compilador

OpenMP permite paralelização através de diretivas como `#pragma omp parallel for`.

## Distribuição de Carga

Loops são distribuídos automaticamente entre threads disponíveis, aproveitando múltiplas cores.

## BLAS Paralelo

Implementação com BLAS otimizado para memória compartilhada e acesso paralelo a dados.

# CUDA: Computação em GPU

## Arquitetura CUDA

Milhares de cores paralelos organizados em blocos e threads para máxima eficiência computacional.

## CUBLAS

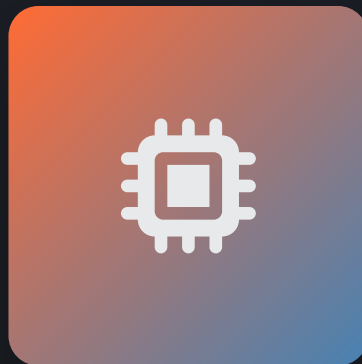
CUDA Basic Linear Algebra Subprograms para operações matriciais otimizadas em GPU.

## Paralelismo Massivo

Ideal para operações matriciais onde cada elemento pode ser calculado independentemente.

## Memória de Alta Velocidade




Acesso rápido à memória compartilhada e uso de técnicas de tiling para otimização.






Processamento  
Paralelo  
Massivo

# Testes de Desempenho: Metodologia

## Metodologia

-  timeit para medições precisas
-  Testes repetidos para confiabilidade
-  Médias estatísticas para análise

## Métricas

-  Tempo de execução
-  Speedup vs Python puro
-  Eficiência de uso de recursos

## Implementações Comparadas

Python puro (loops)



NumPy (vetorizado)



OpenMP (paralelo)



CUDA (GPU)



# Análise de Resultados



## Speedup

Comparação gráfica de tempos e ganhos de performance.



## Custo-Benefício

Análise entre esforço de desenvolvimento e ganho de performance.



## Limitações

Overhead de comunicação e transferência de dados GPU-CPU.



## Recomendações

Diretrizes para diferentes cenários e tamanhos de problema.

## Escala de Speedup Esperado

1x

Python Puro

100x

NumPy

500x

OpenMP

2000x

CUDA



# Implementação Prática: Passo a Passo

## 1 Estrutura do Projeto

Arquivos C/C++ (.c, .cpp, .h)  
Scripts de compilação (Makefile)

- Módulos Python (\_init\_.py)

## 2 Compilação

Flags de otimização (-O3)  
OpenMP (-fopenmp)

- CUDA (nvcc)

## 3 Integração

setup.py para pip  
Wrappers Python

- Documentação e exemplos

```
# Exemplo de compilação gcc -O3 -fopenmp -shared -fPIC matrix_mult.c -o matrix_mult.so nvcc -O3 -shared -fPIC matrix_mult_cuda.cu -o matrix_mult_cuda.so
```



# Conclusões e Próximos Passos

A combinação de CPython com OpenMP e CUDA permite superar limitações nativas do Python, alcançando desempenho de alto nível mantendo a simplicidade da linguagem. A abordagem híbrida é essencial para aplicações de grande porte em ciência de dados.



Fork no GitHub



Documentação



Comunidade