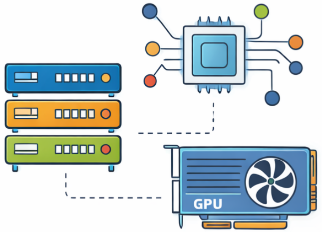
High-Performance Computing with Python

Abolfazl Ziaeemehr

Institut de Neuroscience des Systèmes, Inserm, Aix-Marseille University

January 27, 2026



Outline

Why High-Performance Python?

- Python is expressive, readable, and productive
- But naïve Python can be slow
- HPC Python = **identify bottlenecks + use the right tool**

Key message:

Do not optimize blindly. Measure first.

Profiling: The First Step

- Performance intuition is often wrong
- Profiling tells you:
  - Where time is spent
  - Which functions dominate runtime
- Optimization without profiling is guesswork

Types of Profiling

- **Time profiling**
  - CPU time per function
  - Line-by-line cost
- **Memory profiling**
  - Allocation hotspots
  - Leaks and excessive copies

Notebook Demo

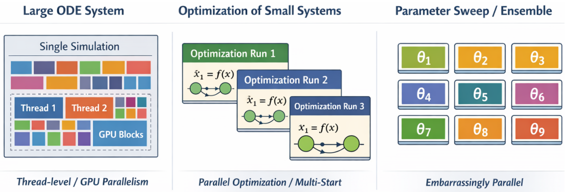
Live demo

Profiling notebooks:

- cProfile / line_profiler
- Memory profiler

Different problems require different parallelization approaches:

- **Large ODE systems** (e.g., 10,000+ equations)
  - Multi-threading within solver
  - Parallel linear algebra operations
  - GPU acceleration (CUDA, OpenCL)
- **Small ODE systems with optimization**
  - Parallelize the optimization algorithm
  - Multiple optimization runs simultaneously
  - Parallel gradient computations
- **Parameter sweeps / Ensemble simulations**
  - Embarrassingly parallel
  - Each parameter set runs independently
  - Ideal for multiprocessing or job arrays



Example: ODE Parallelization Strategies

Within-solver parallelism:

- Jacobian computation
- Matrix operations
- Implicit method solves

Across-simulation parallelism:

- Different initial conditions
- Parameter space exploration
- Monte Carlo simulations

Key message:

Choose your parallelization strategy based on the problem structure, not just the tools available.

Multiprocessing in Python

- Python has a Global Interpreter Lock (GIL)
- Threads do not run Python bytecode in parallel
- **Multiprocessing** uses multiple OS processes

When Multiprocessing Works Well

- CPU-bound tasks
- Embarrassingly parallel problems
- Independent simulations / parameter sweeps

Examples:

- Batch data processing

Costs of Multiprocessing

- Process startup overhead
- Data serialization (pickling)
- Memory duplication

Rule of thumb

Large tasks, coarse parallelism.

Notebook Demo

Live demo

Multiprocessing notebooks:

- multiprocessing.Pool
- Joblib

Numba: JIT Compilation

- Just-In-Time compilation for Python
- Compiles numerical code to machine code
- Minimal code changes

What Numba Excels At

- Tight loops
- Numerical kernels
- Numpy-like code with explicit loops

Mental model

Numba turns Python loops into C-like speed.

Limitations of Numba

- Limited Python features
- Dynamic typing is restricted
- Compilation overhead on first call

Notebook Demo

Live demo

Numba notebooks:

- @njit
- Parallel loops

CuPy: Numpy on the GPU

- GPU-accelerated array library
- Numpy-compatible API
- CUDA backend

When CuPy Is a Good Fit

- Large array operations
- Linear algebra
- Elementwise kernels

Key idea

Same code style, different device.

Costs and Pitfalls

- Data transfer CPU $\leftrightarrow$ GPU
- GPU memory limits
- Small arrays often slower

Notebook Demo

Live demo

CuPy notebooks:

- cupy.asarray
- GPU speedups

JAX: Composable High-Performance Python

- Functional programming style
- XLA compilation
- CPU, GPU, TPU support

Core JAX Transformations

- **jit** – compilation
- **vmap** – vectorization
- **grad** – automatic differentiation

JAX RNG: Stateless Random Generation

- Functional approach to randomness
- PRNG keys for reproducibility
- Safe for jit and vmap

Why JAX Is Powerful

- Performance + composability
- Differentiable programming
- Research-grade and production-grade

Advanced JAX Concepts

- **PyTree**: Nested data structures
- **LAX**: Low-level operations
- **Scan**: Efficient loops

Trade-offs

- Functional mindset required
- Less dynamic than pure Python
- Compilation latency

Notebook Demo

Live demo

JAX notebooks:

- jit vs pure Python
- vmap patterns
- Random number generation
- Advanced concepts (PyTree, LAX, Scan)

SWIG: Python Meets C/C++

- Interface generator
- Wrap existing C/C++ code
- Use legacy HPC libraries

When SWIG Makes Sense

- Existing C/C++ codebase
- Performance-critical kernels
- Long-term maintenance

Choosing the Right Tool

Problem	**Tool**
Unknown bottleneck	Profiling
CPU-bound loops	Numba
Parallel tasks	Multiprocessing
GPU arrays	CuPy
Composable HPC	JAX
Legacy C/C++	SWIG

Final Message

High-performance Python is not one tool — it is knowing **when** to use each tool.

Questions?