Python performance Past, Present, Future

Victor Stinner

EuroPython 2019, Basel

Red Hat


Past

Python Implementations

- 1989: CPython (Guido van Rossum)
- 1997: Jython (Jim Hugunin)
- 1998: Stackless Python (Christian Tismer)
- 2006: IronPython (Jim Hugunin)
- 2014: MicroPython (Damien George)

Faster Pythons

- 2002-2012: psyco (Armin Rigo)
- 2007: PyPy
- 2009-2010: Unladen Swallow (Google)
- 2014-2017: Pyston (Dropbox)
- 2016-2017: Pyjion (Microsoft)

Notice end date for most of these projects...

Two approaches

- Fork CPython
- Implementation from scratch

Fork CPython

- Unladen Swallow, Pyston, Pyjion, ...
- Performance limited by **old CPython design** (1989)
- Specific memory allocators, C **structures**, reference counting, specific GC, ...
- CPython is limited to **1 thread** because of the **GIL** (more later)

From scratch

- PyPy, Jython, IronPython, ...
- Jython and IronPython have **no GIL**!
- PyPy uses an efficient **tracing garbage collector**
- **C extensions**: no support, or slow
- cpyext creates CPython PyObject on demand, sync with PyPy objects

Competition with CPython

- CPython has around **30 active core developers** to maintain it
- **New features** first land into CPython
- Why would users prefer an outdated and incompatible implementation?
- Who will sponsor the development?

Unladen Swallow (2011)

- “Most **Python** code at Google **isn't performance critical**”
- “**deployment** was too difficult: being a replacement was not enough.”
- “Our potential customers eventually **found other ways** of solving their performance problems.”

http://qinsb.blogspot.com/2011/03/unladen-swallow-retrospective.html

Pyston & Dropbox (2017)

- “Dropbox has increasingly been writing its **performance-sensitive** code in other languages, such as **Go**”
- “We spent much more time than we expected on **compatibility**”

https://blog.pyston.org/2017/01/31/pyston-0-6-1-released-and-future-plans/

Summary

- CPython remains the **reference implementation** but shows its age
- Multiple optimization projects **failed**
- PyPy: **drop-in replacement**, 4.4x faster, but not widely adopted yet: **why?**


Present

Optimize your code

- Let’s say that you identified your code causing the performance bottleneck
- How to make your code faster?

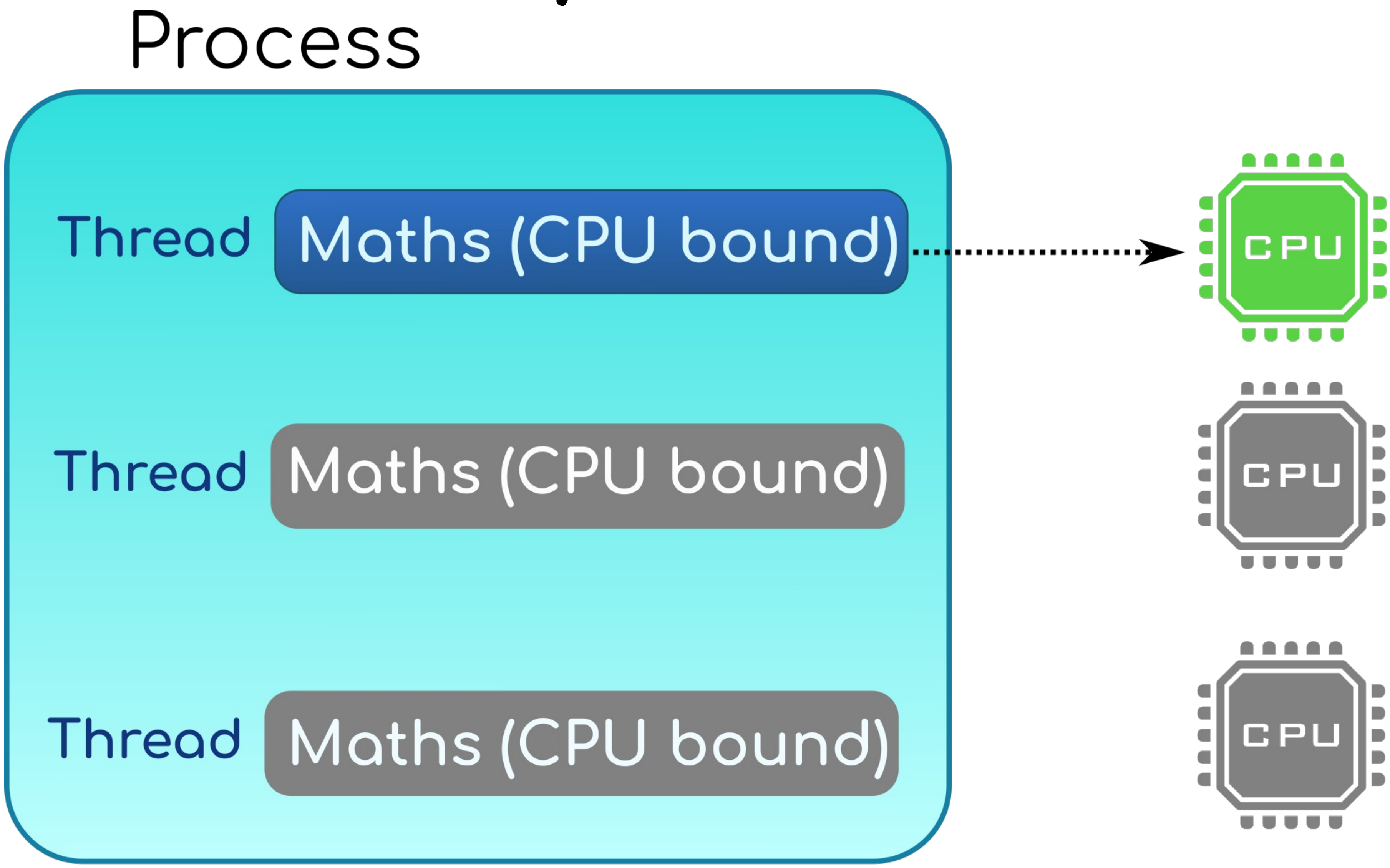
PyPy just works!

- Drop-in replacement for CPython!
- 4.4x faster than CPython in average (exact speedup depends on your workflow)
- Fully compatible with CPython

PyPy issues

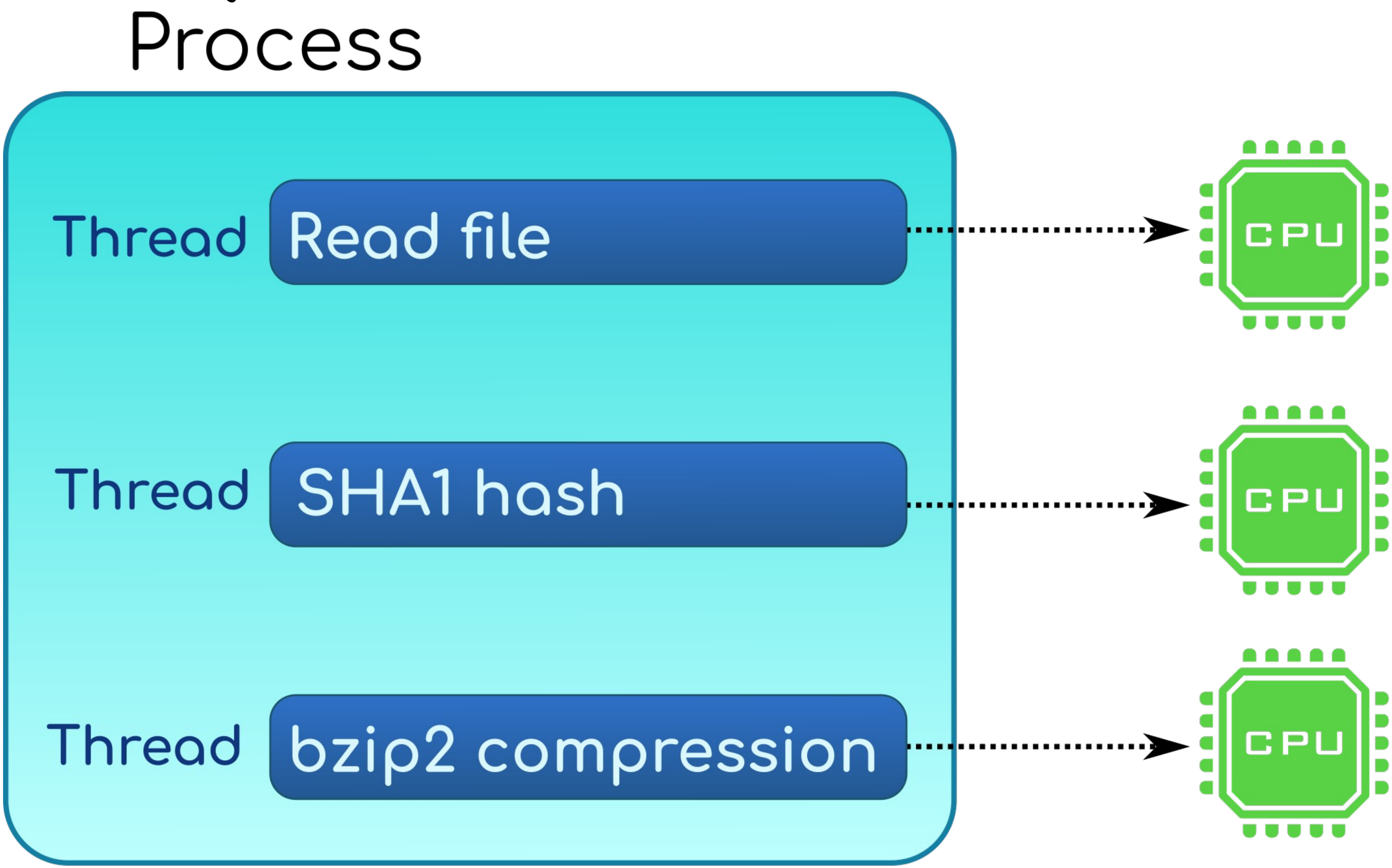
- Support C extension with cpyext: heavily optimized in 2018, but still slower than CPython
  (more about the C API later...)
- Larger memory footprint (JIT)
- Longer startup time (JIT)

CPython GIL



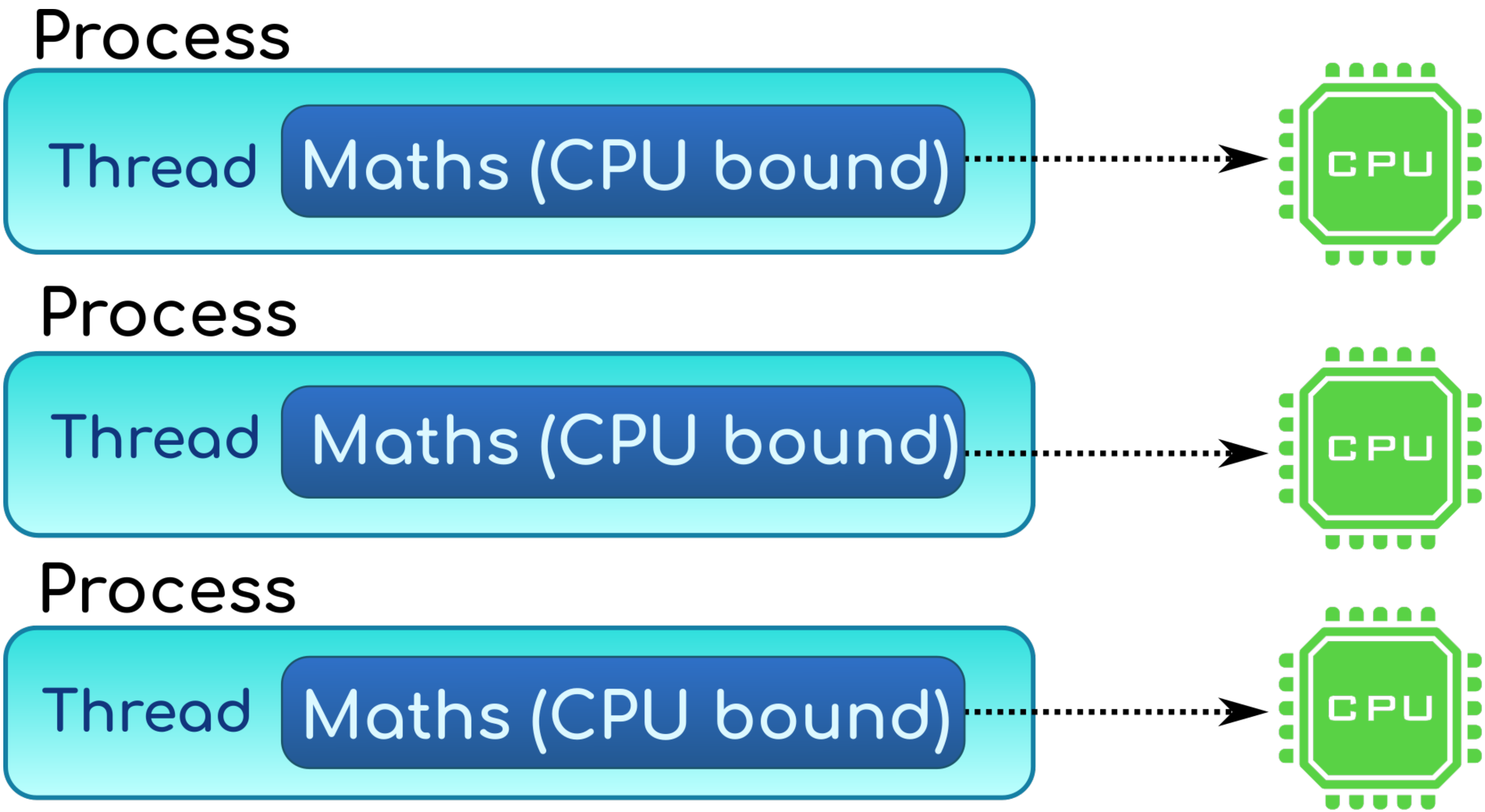
Efficiency: 1/3

CPython: release the GIL



Efficiency: 100%

CPython: multiprocessing



Efficiency: 100%

Multiprocessing

- Work around the **GIL** limitation
- **Shared memory** (Python 3.8) avoids memory copies between workers
- New **pickle** version 5 (Python 3.8) **avoids copying** large objects: PEP 574

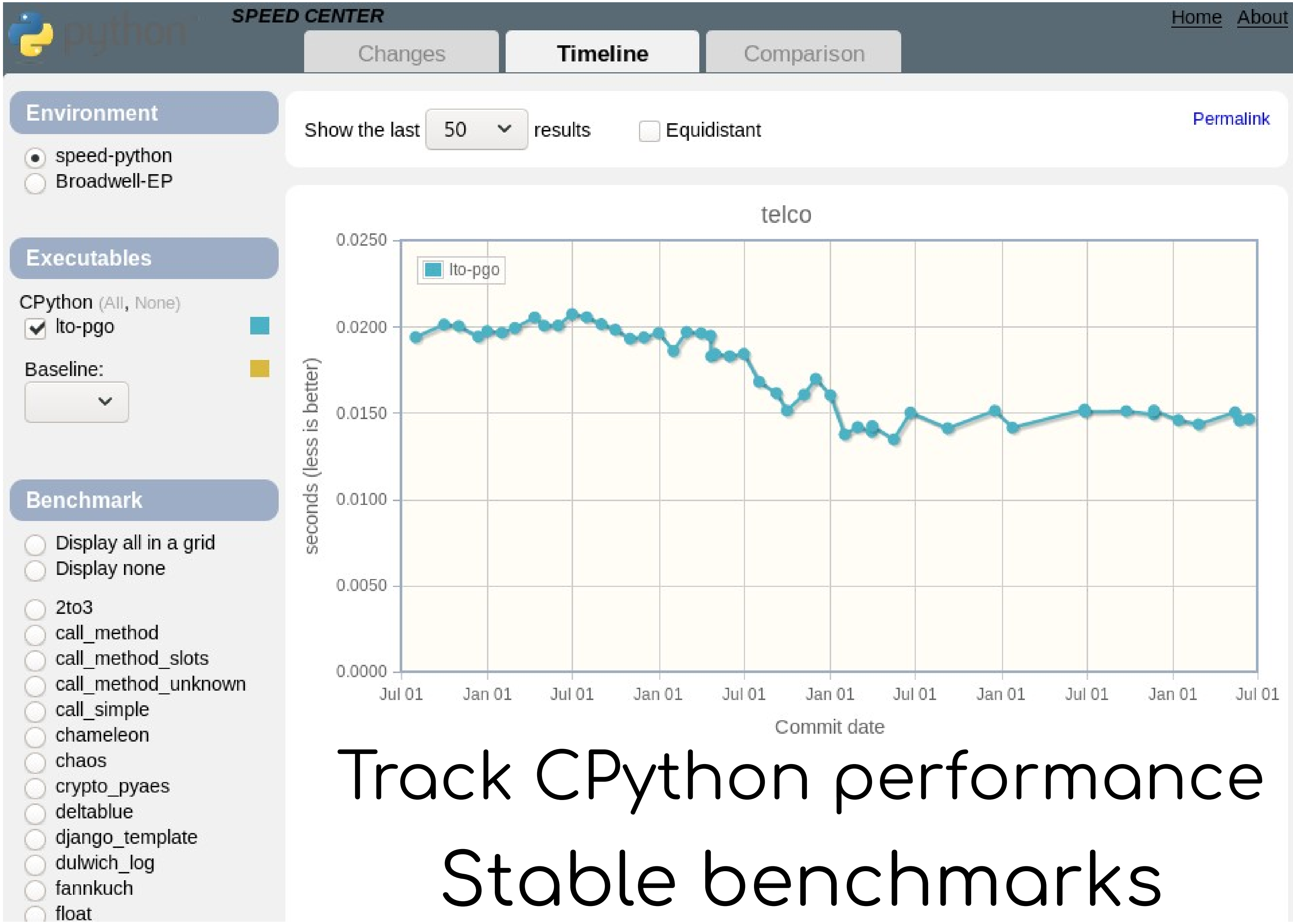
Cython

- Easy way to write C extension
- Syntax similar to Python
- Support **multiple** Python **versions**
- **Handle reference counting** for you
- The optimizer emits efficient code using CPython internals for you

Numba

- JIT compiler translating subset of Python and NumPy into fast code
- Simplified Threading: release GIL
- SIMD Vectorization: SSE, AVX, AVX-512
- GPU Acceleration: NVIDIA's CUDA and AMD's ROCm

https://speed.python.org



Track CPython performance

Stable benchmarks

Summary

- **PyPy** doesn't require any code change
- **Multiprocessing** scales
- Use **Cython**, don't use the C API directly
- **Numba** makes numpy faster


Future

Python C API

Python C API

- Evolved organically: internal functions are **exported by mistake**
- First written to be consumed by CPython itself
- **No design**: expose everything
- It exposes too many **implementation details**

Python 3.8 changes

Python 3.8 now has 3 levels of C API:

- Include/: public “portable” C API
- Include/cpython/: C API specific to CPython
- Include/internal/: internal C API

Many private functions (“_Py...”) and PyInterpreterState structure moved to the internal API.

Stable ABI

- Support **multiple** Python **versions**: Python 3.8, 3.9, ...
- CPython 3.8 **debug build** is **ABI compatible** with the release build
- It can use C extensions compiled in release mode
- It has much more sanity checks at runtime to **detect bugs**

Specialized lists

- CPython list: array of **pointers**
  PyObject*
- PyPy specialized list: list of **integers**
  int64_t array[n]
- Can it be implemented in CPython?
- Can we modify PyListObject?

Accessing structs

- Problem 1: PyList_GET_ITEM() macro access directly PyListObject**.ob_item**[index] (PyObject*)
- C extensions must not access PyListObject fields directly
- PyList_GET_ITEM() macro could be modified to convert int64_t to PyObject*, but...

Borrowed reference

- Problem 2: PyList_GET_ITEM() returns a borrowed reference, Py_DECREF() must not be called
- If PyList_GET_ITEM() would create a **temporary object**, **when should it be destroyed**? We don't know...
- Many C functions return borrowed references

Better C API

- Make **structures opaque**. Such code must fail with a compiler error:
  ```
  PyObject *obj = PyLong_FromLong(1);
  PyTypeObject *type = obj->ob_type;
  ```
- **Remove** functions using **borrowed references** or “stealing” refererences
- Replace macros with **function calls**

Break compatibility?

- Any C API change can **break an unknown number of projects**
- Maybe not all C API design issues can be fixed
- **Updating all C extensions** on PyPI will take a lot of time

... there should be another way...

New PyHandle C API

- New C API: correct since day 1
- PyHandle: opaque integer
- Similar to an Unix file descriptor or Windows HANDLE:
  open(), dup(), close()

New PyHandle C API

- CPython: Implemented on top of the current C API
- PyPy: More **efficient** than the current C API
- Cython: **no need to change your code**, Cython will generate code using PyHandle for you

Reference counting

Gilectomy

- "Remove the GIL": replace unique GIL with one lock per mutable object
- Atomic increment/decrement
- Log INCREF/DECREF
- Reference counting doesn't scale with the number of threads

Tracing GC for CPython

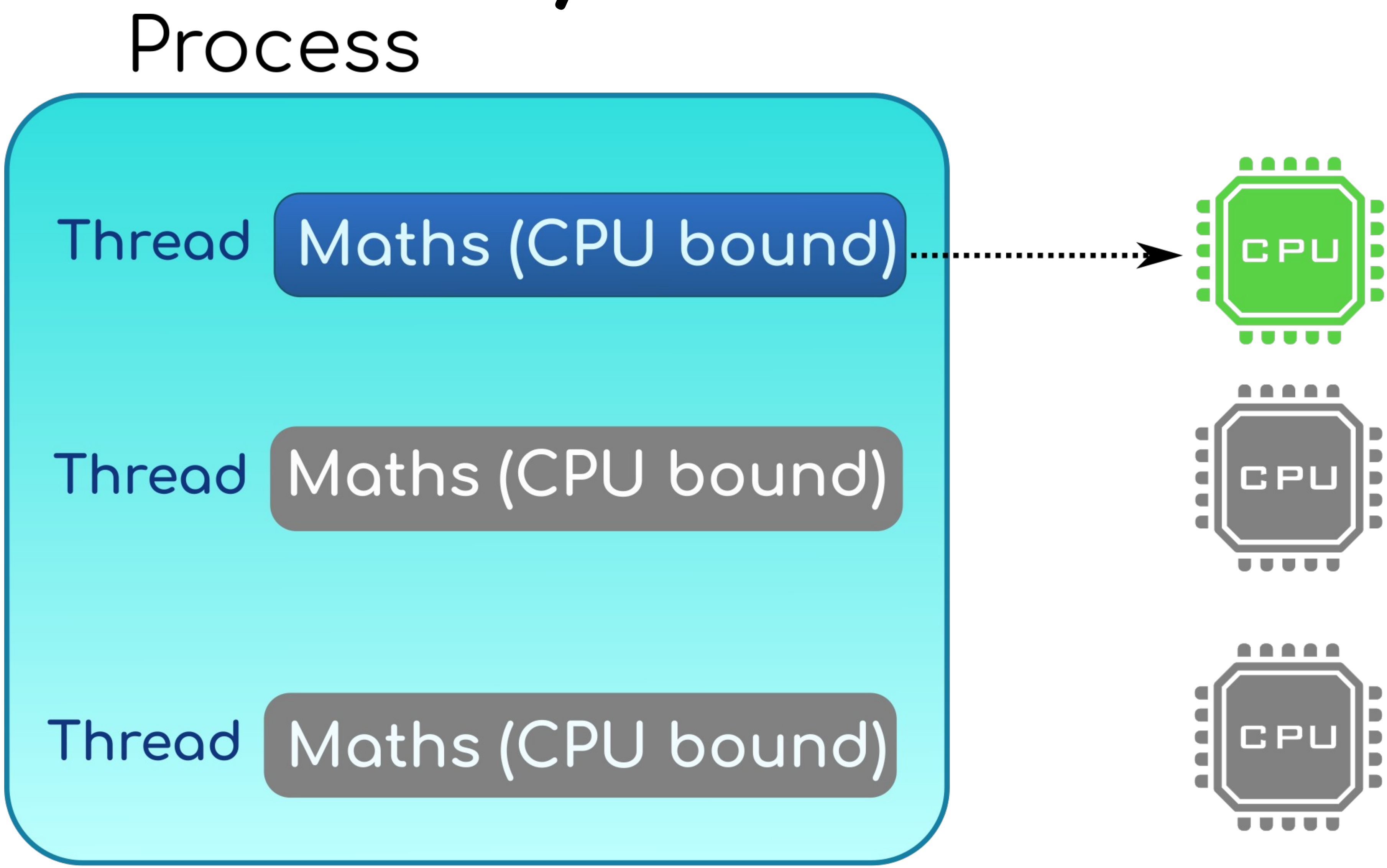
- Many modern language implementations use tracing GC
- PyPy has a tracing GC
- Existing C API would continue to use reference counting

Subinterpreters

Subinterpreters

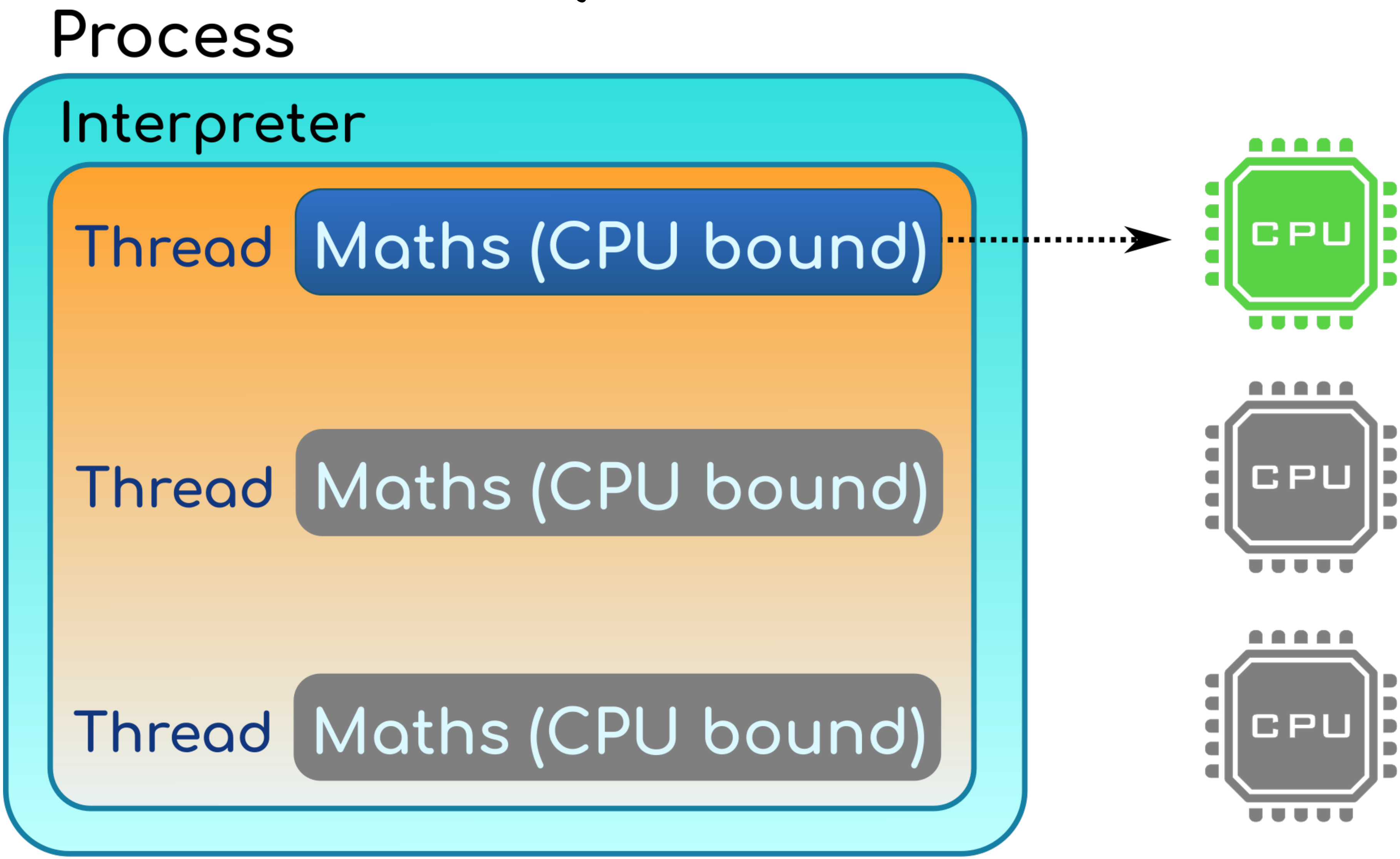
- Eric Snow's PEP 554 "Multiple Interpreters in the Stdlib"
- Replace the unique Global Interpreter Lock (GIL) with **one lock per interpreter**
- **Work-in-progress** refactoring of CPython complex internals

CPython GIL



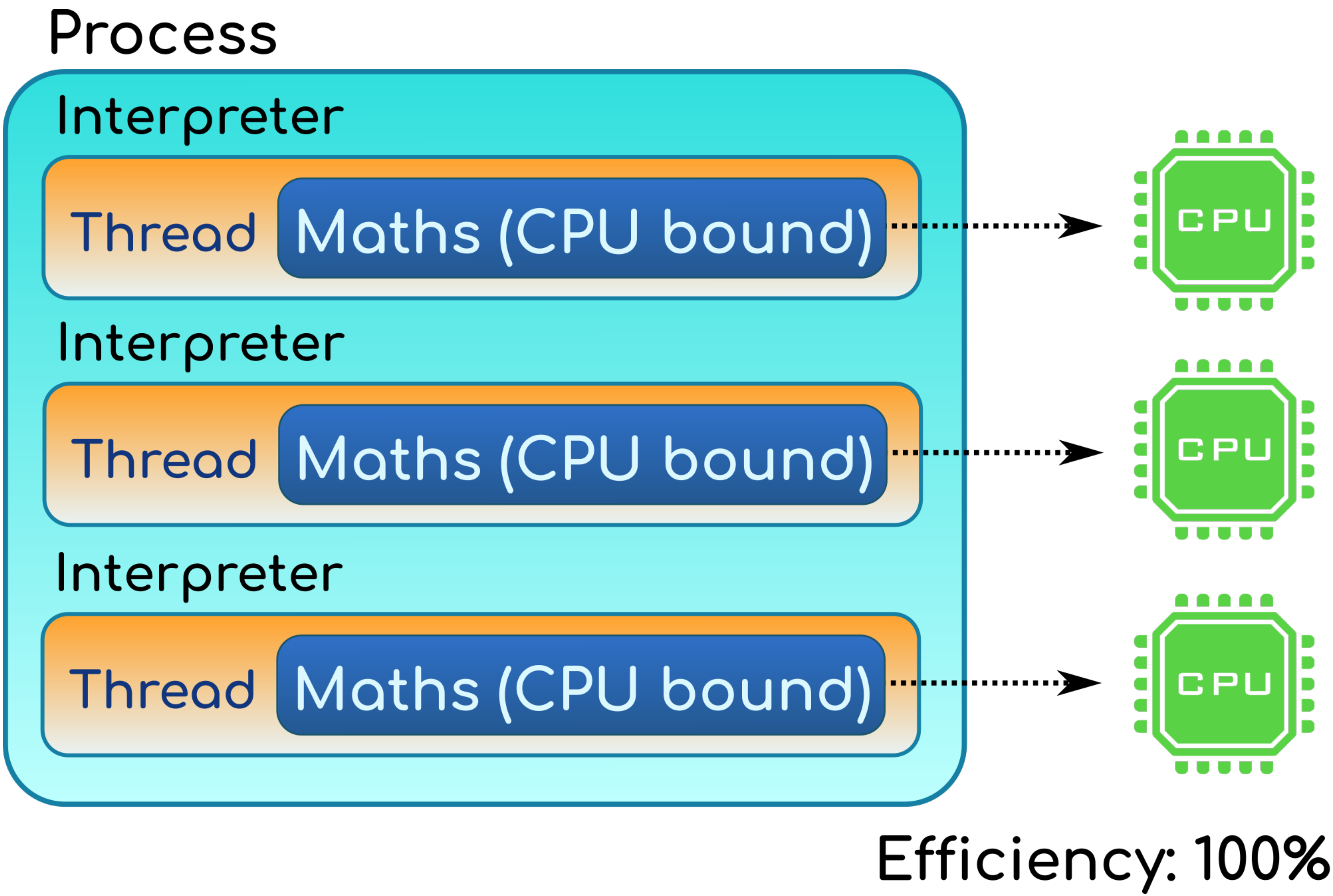
Efficiency: 1/3

CPython GIL



Efficiency: 1/3

CPython subinterpreters



Subinterpreters

Expectation (should be verified with a working implementation):

- **Lower memory footprint**: share more memory
- **Faster locks** (no syscall?)
- Limitation: Python objects cannot be shared between interpreters

Summary

- Current C API has **design issues**
- New "**PyHandle**" C API
- **Tracing garbage collector** for CPython
- CPython **subinterpreters**

Conclusion

Conclusion

- Many previous optimizations projects failed
- **Cython**, **multiprocessing** and **Numba** are working well to make Python faster
- PyHandle, tracing GC and subinterpreters are very promising!

Questions?

- http://pypy.org/
- https://faster-cpython.readthedocs.io/
- https://pythoncapi.readthedocs.io/
- https://speed.python.org/
- https://mail.python.org/mailman3/lists/speed.python.org/