

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- Why would users prefer an outdated and incompatible implementation?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



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

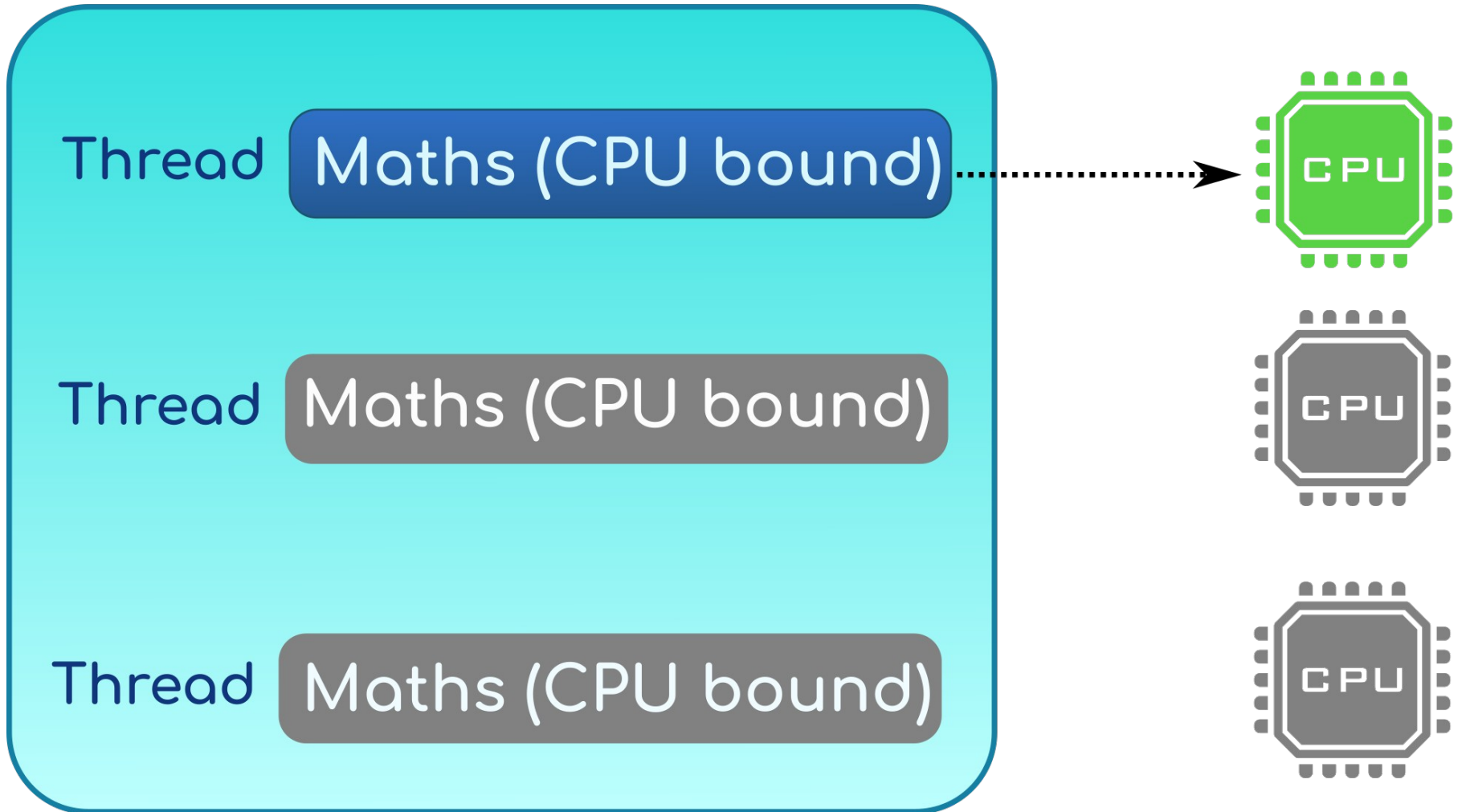


- Subtle behavior differences with CPython on corner cases, ex: PyPy calls `__del__()` “later”
- Support C extension with **cpyext**: slower than CPython, heavily optimized in 2018
- Larger **memory footprint** (JIT)
- Longer **startup time** (JIT)



CPython GIL

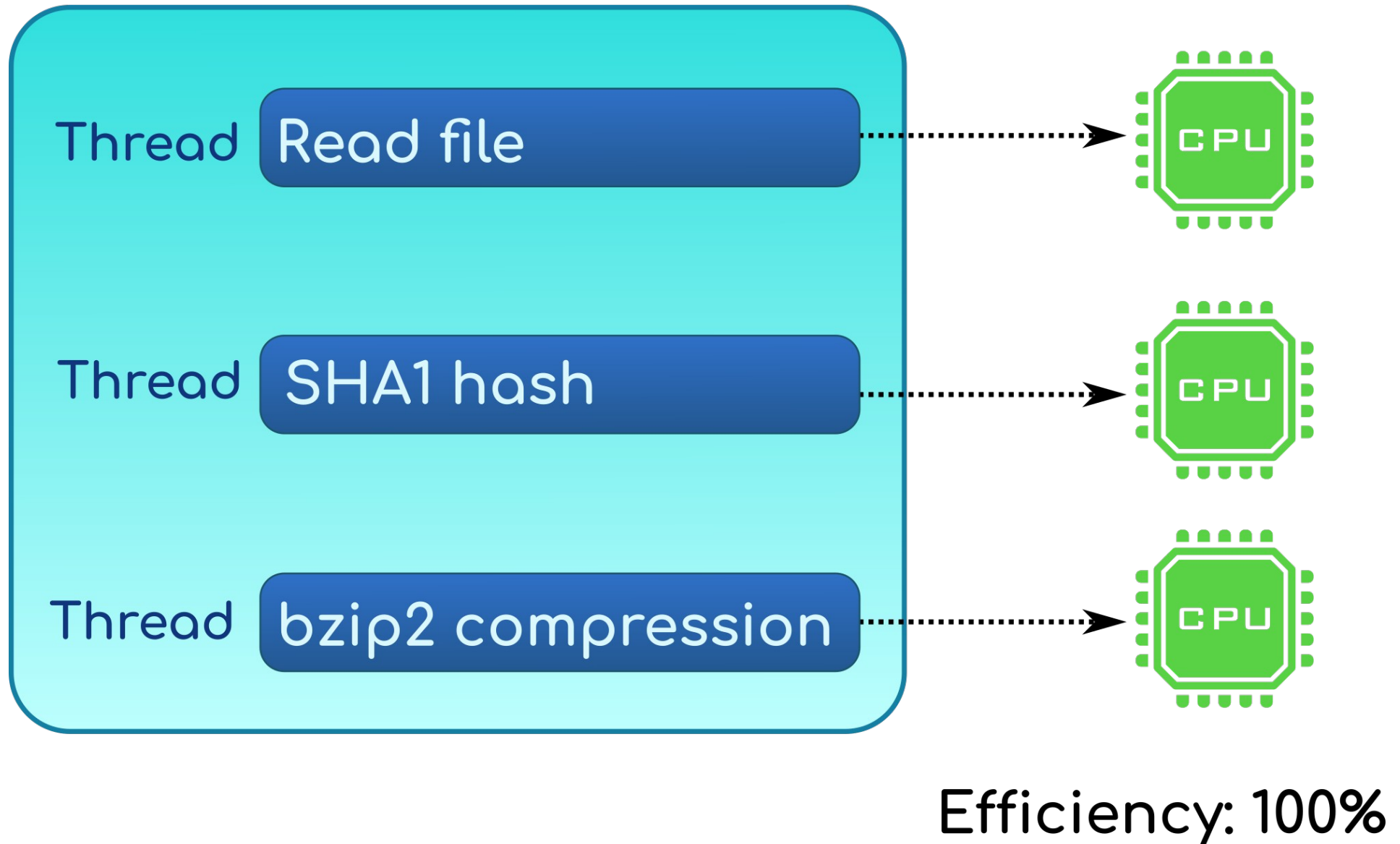
Process



Efficiency: 1/3

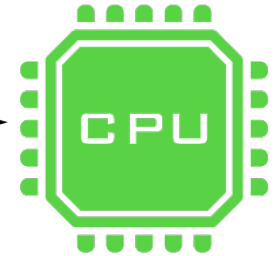
CPython: release the GIL

Process

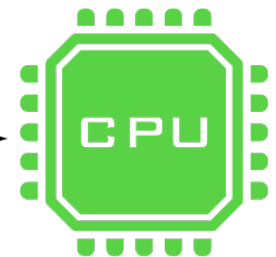


CPython: multiprocessing

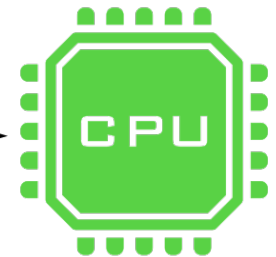
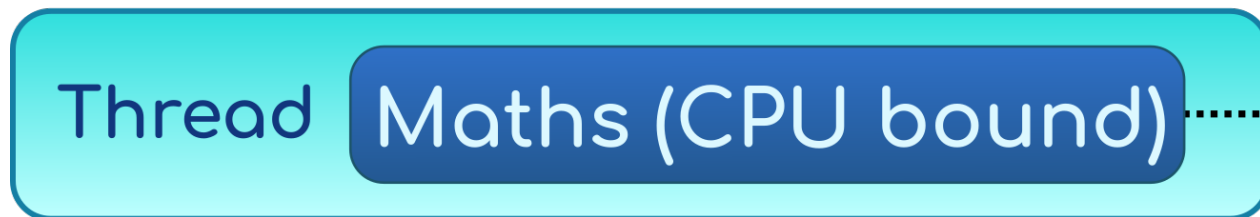
Process



Process



Process



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



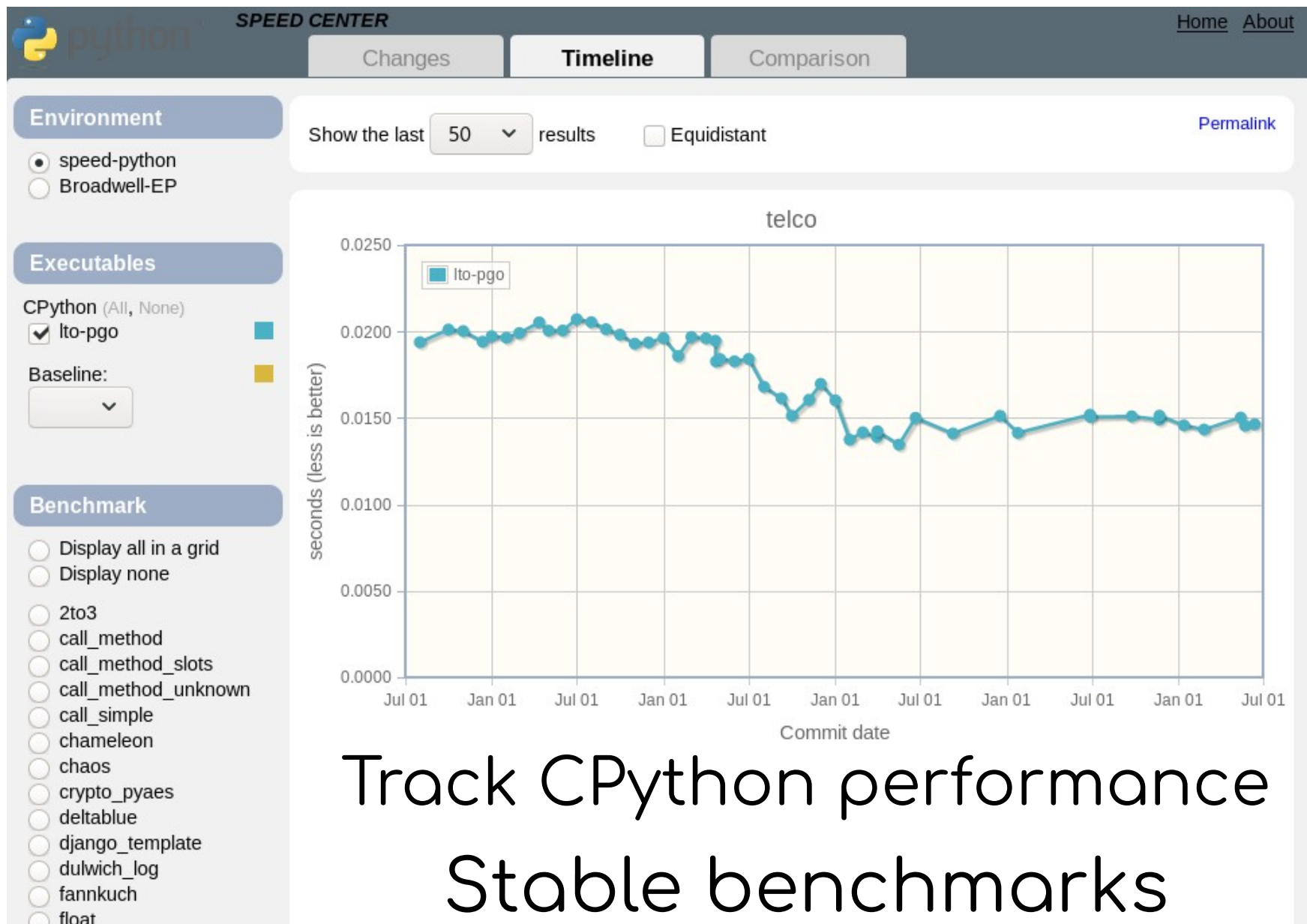
Numba



- **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>



Summary



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





Future

Python C API



- Evolved organically: internal functions are **exported by mistake**
- **No design**: expose everything
- API exposes **implementation details**: structure fields, memory allocators, reference counting, ...



Python 3.8 changes



Python 3.8 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...”) moved to internal/



Stable ABI



- Support **multiple** Python **versions**:
Python 3.8, 3.9, ...
- CPython 3.8 **debug build** is **ABI compatible** with the release build
- Debug build can use C extensions compiled in release mode



Accessing structs



- Python list: array of **pointers** (PyObject*)
- PyPy specialized list: list of **integers** (in C: int64_t array[n])
- Problem 1: PyList_GET_ITEM() macro access directly
PyListObject.**ob_item**[index]
- C extensions must not access PyListObject directly



Borrowed reference



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



Better C API



- Make structures opaque. Must fail with a compiler error:
`PyObject *obj = PyLong_FromLong(1);`
`PyObject *type = obj->ob_type;`
- **Remove** functions using **borrowed references** or “stealing” references.
- 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
- Implemented on top of CPython C API
- **Efficient on PyPy**
- Cython: **no need to change your code**



Subinterpreters

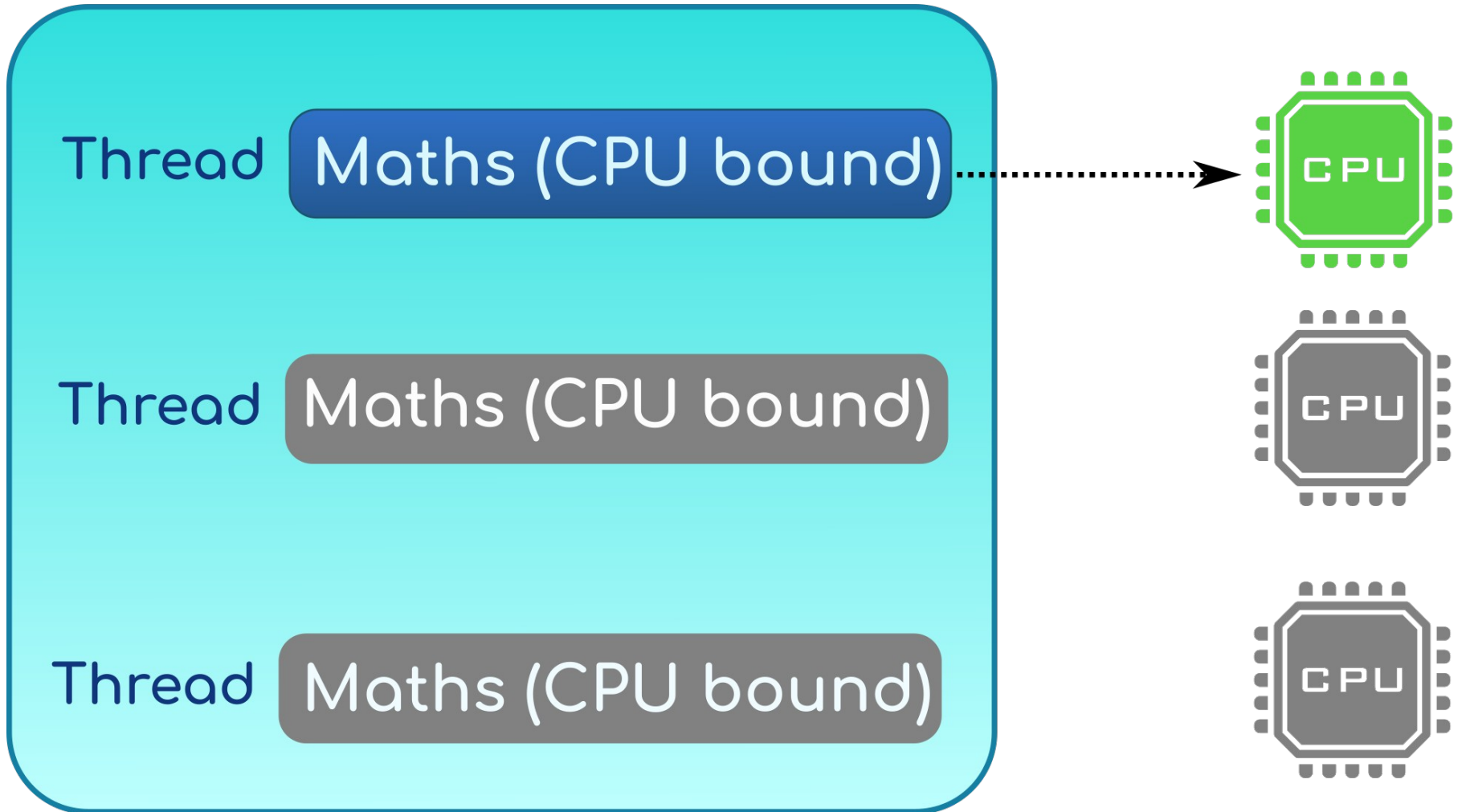


- Eric Snow's PEP 554 "Multiple Interpreters in the Stdlib"
- Replace the unique Global Interpreter Lock (GIL) with **one lock per interpreter**
- Work-in-progress refactoring of CPython complex internals



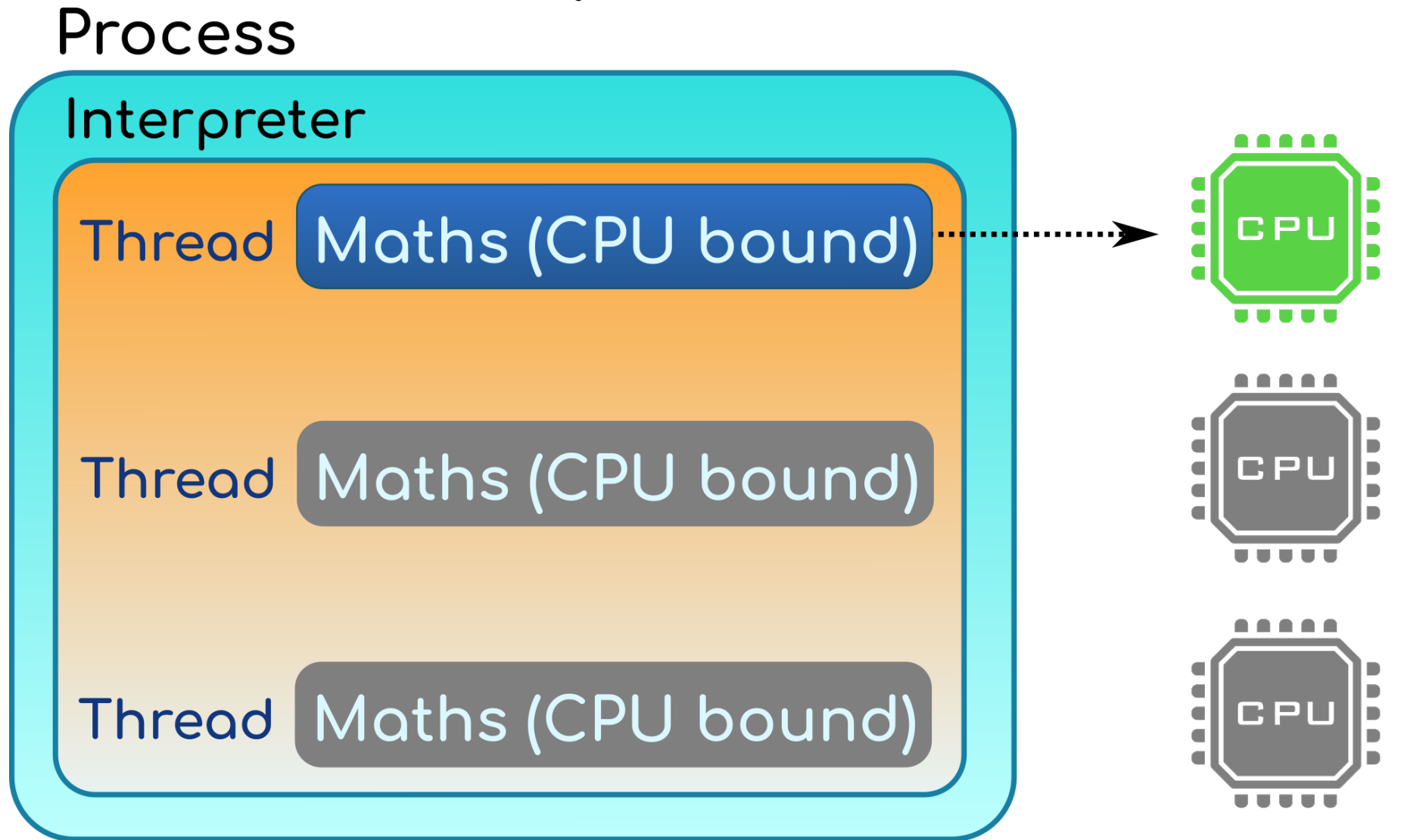
CPython GIL

Process



Efficiency: 1/3

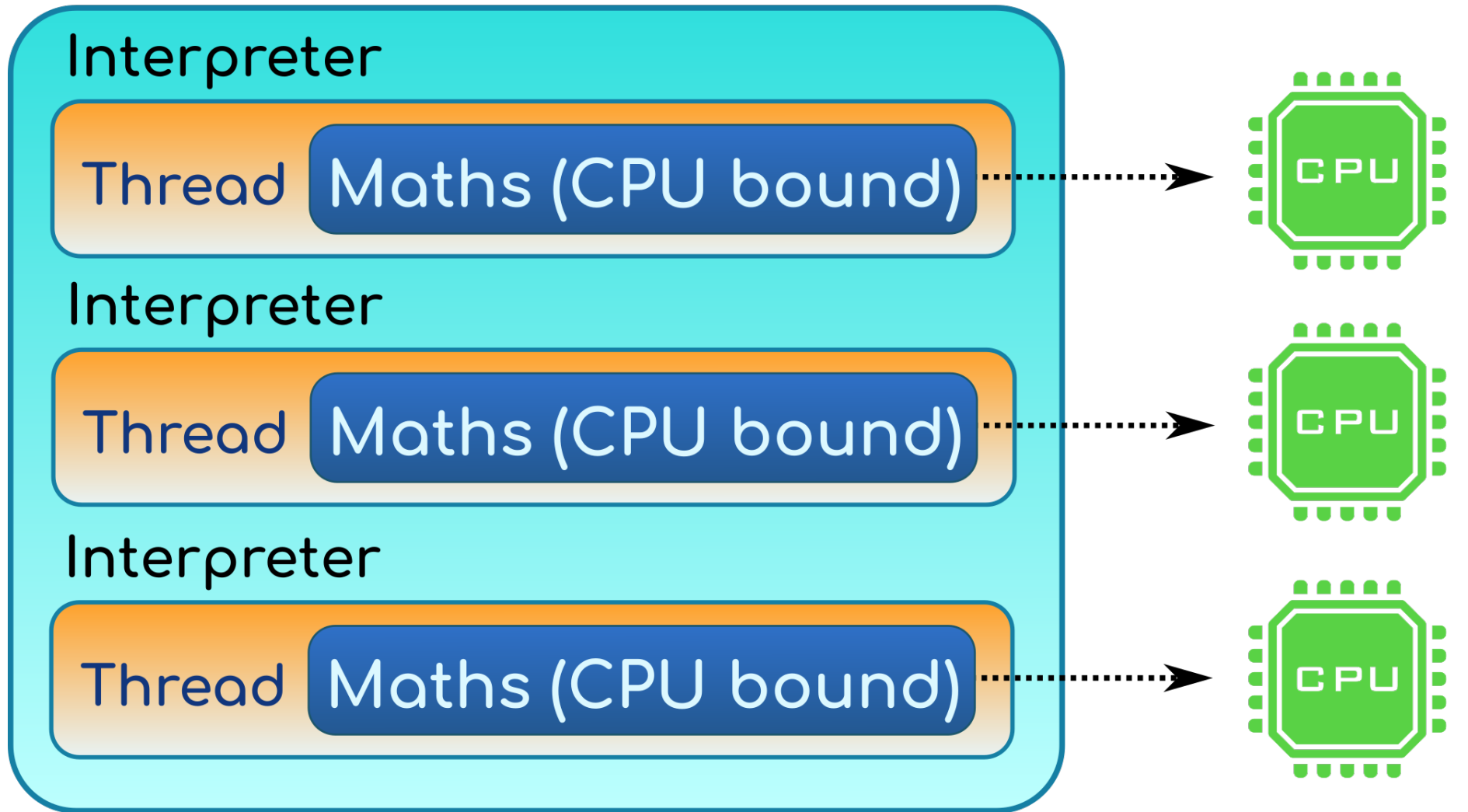
CPython GIL



Efficiency: 1/3

CPython subinterpreters

Process



Efficiency: 100%

Subinterpreters



Expectation (should be verified with a working implementation):

- **Lower memory footprint:** share more memory
- **Faster locks**
- Limitation: Python objects cannot be shared between interpreters.



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**



Summary



- Enhance existing C API
- Write a new “PyHandle” C API
- CPython subinterpreters
- Tracing garbage collector for CPython



Conclusion



Conclusion



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

