HPy

The Future of Python Native Extensions

Florian Angerer
Štěpán Šindelář

Oracle Labs
GraalPy and HPy Core Developers

About Us and This Talk

- Who are we
  - Florian Angerer and Štěpán Šindelář
  - Software Engineers @ Oracle Labs
  - GraalPy core developers https://graalvm.org/python
  - HPy core developers https://hpyproject.org
    - not in group of HPy founders but joined very early in 2019
- This Talk
  - Motivates and introduces HPy
  - Shows benefits to you
  - Convince you to use HPy ☺

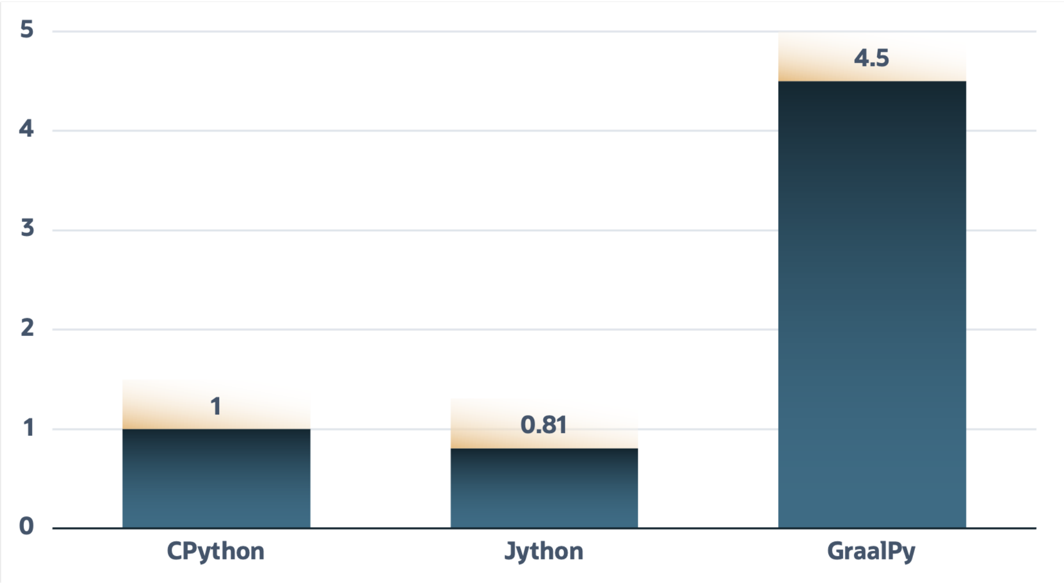
Target Audience

- “Experienced” Python programmer
  - Maybe even Python package developer
- You know C or the C memory model (pointers, struct, malloc, …)
- Have a vague understanding of the (C)Python internals
- Have a vague understanding of `#include <Python.h>`

CPython and Alternative Runtimes

- CPython
  - The Python reference implementation
  - A bytecode interpreter
  - Written in C
- Alternative Runtimes
  - GraalPy, PyPy, Jython, IronPython, Pyston, Cinder, …
- Most of them try to improve Python language execution speed by using a just-in-time compiler, different data structures, moving GC, etc.

On average, GraalPy is 4.5x faster than CPython.



Geomean speedup over CPython on the Python Performance suite
(Note that Jython can only run a subset of the benchmarks due to the missing Python 3 support)

Python C API

- Since CPython is written in C
- Allow C extensions
- Exposes (internal) APIs, data structures, and implementation details
  - `ob_refcnt`, `ob_type`, `PyTypeObject`, …
- Reference counting
- Objects as C pointers (`PyObject *`)
  - Exposes memory location
  - Assumption about identity
- Borrowed references
- Victor Stinner’s PEP 620 https://peps.python.org/pep-0620/

Example: Reference Counting vs. GC

- GraalPy is written in Java
- Java has the most advanced and mature GC implementations
- Reference counting prevents using a “real” GC
- The GC is NOT (only) about collecting garbage!
  - It should be called Memory Manager
- State-of-the-art GCs
  - super-fast allocation, fast deallocation
  - no/minimal pauses
  - multi-threading
- Why should I care about that?

~79x faster

GraalPy compared to CPython on gcbench.py

PyPy is also ~25x faster than CPython

What is HPy?

- A novel C API for writing Python extensions
    - #include “hpy.h” instead of “Python.h”
- Funded directly/indirectly via OpenCollective, Oracle, IBM, Quansight Labs, and Anaconda Inc.
- Mostly driven by PyPy and GraalPy developers
- HPy: more abstract
    - Hides implementation details
    - Easier/faster on alternative implementations
    - GC friendly

HPy Goals

- Zero overhead on CPython
  - Using macros and static inline functions to map HPy calls to C API calls
- Incremental migration path
  - Port existing extensions one function at a time
- Faster on alternative implementation
  - PyPy, GraalPy, …
- Better debugging experience
- Universal ABI
  - One binary for multiple Python interpreters
- Backwards Compatibility
  - Multiple ABIs in the same interpreter

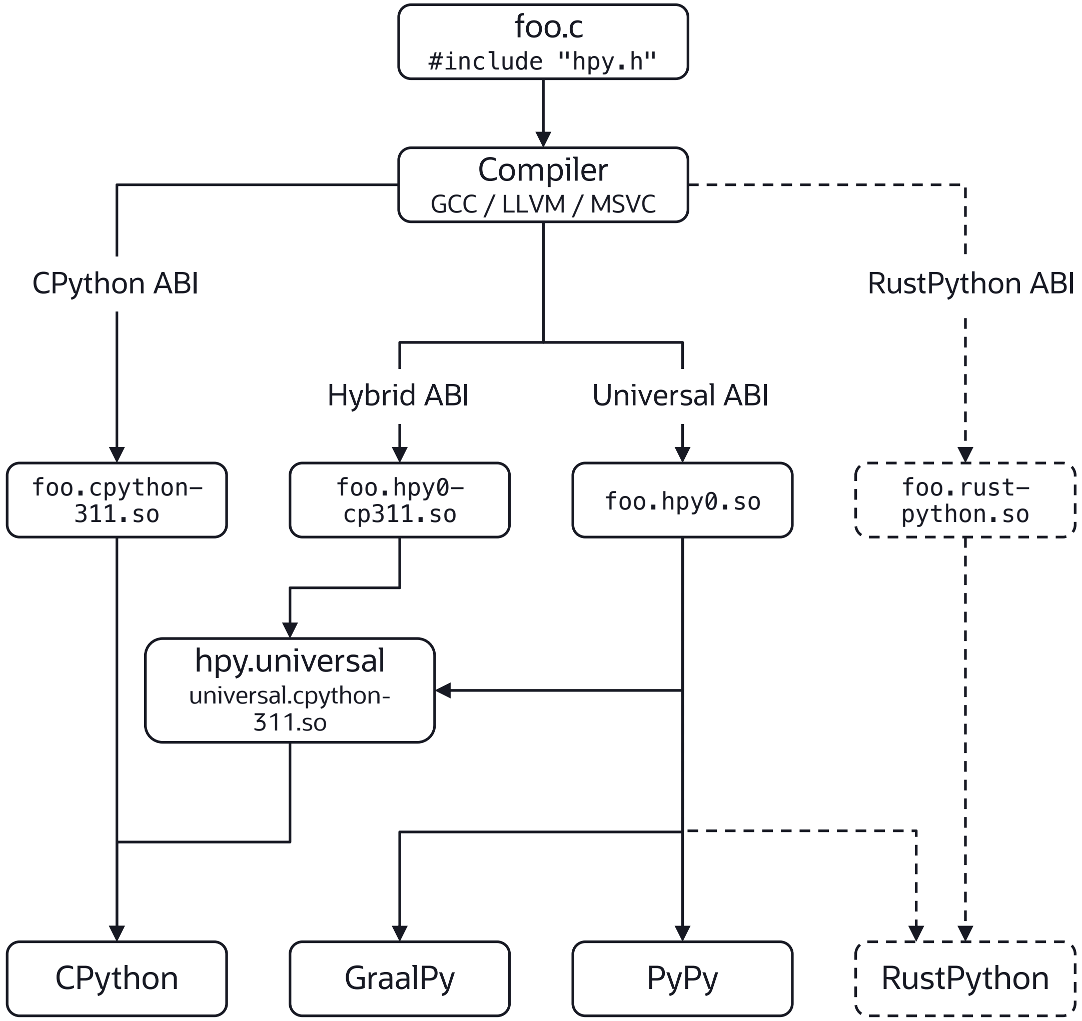
How Does HPy Look?

```c
// quickstart.c

#include "hpy.h" // instead of "Python.h"

HPyDef_METH(say_hello, "say_hello", HPyFunc_NOARGS)
static HPy say_hello_impl(HPyContext *ctx, HPy self)
{
    return HPyUnicode_FromString(ctx, "Hello world");
}
static HPyDef *QuickstartMethods[] = {
    &say_hello,
    NULL,
};
static HPyModuleDef quickstart_def = {
    .doc = "HPy Quickstart Example",
    .defines = QuickstartMethods,
};

HPy_MODINIT(quickstart, quickstart_def)
```

```python
# setup.py
#
# python3 setup.py [--hpy-abi=universal] build

from setuptools import setup, Extension
from os import path

DIR = path.dirname(__file__)
setup(
    name='hpy-quickstart',
    hpy_ext_modules=[
        Extension('quickstart’,
                  sources=[path.join(DIR, 'quickstart.c')]),
    ],
    setup_requires=['hpy'],
)
```

Zero Overhead on CPython

- Multiple compilation modes
- Universal ABI
    - Common ABI supported by all interpreters
    - One binary for all interpreters
    - API calls are done via `HPyContext`
- Custom ABIs
    - Maps HPy API functions to interpreter-specific APIs
    - E.g. CPython ABI
    - Binary is interpreter-specific
    - Best performance for target interpreter
- Hybrid ABI



Incremental Migration Path

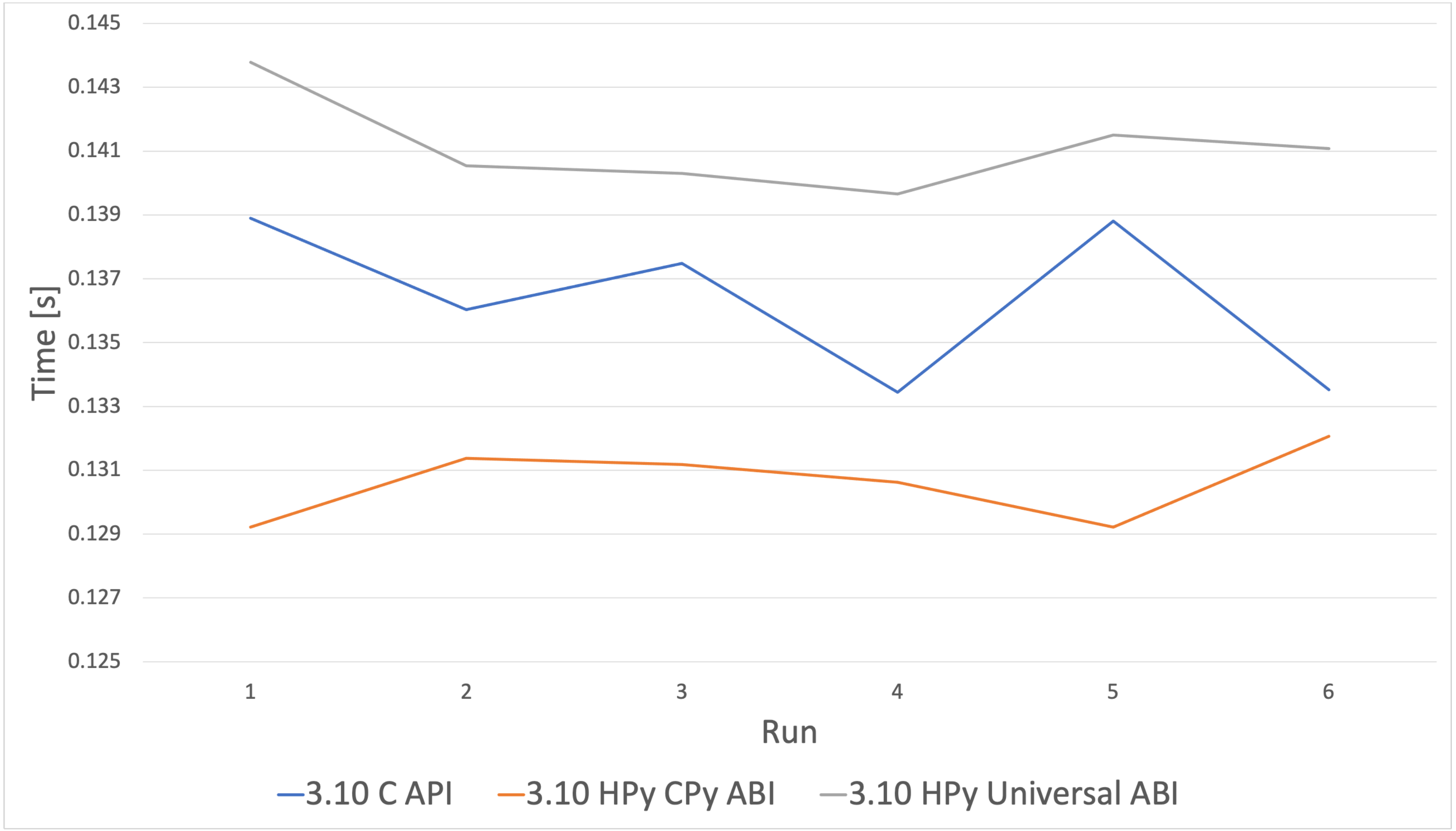
- Convert to HPy module
  - Keep existing functions as legacy methods
- Migrate types to (legacy) HPy type
- Migrate functions, slots, and members one by one
- Use `HPy_AsPyObject` and `HPy_FromPyObject`
- Build and test after each step with `--hpy-abi=hybrid`

```
static PyMethodDef PointModuleLegacyMethods[] = {
    {"dot", (PyCFunction)dot, METH_VARARGS, "Dot product."},
    {NULL, NULL, 0, NULL}
};

static HPyDef *module_defines[] = {
    &module_exec,
    NULL
};

static HPyModuleDef moduledef = {
    .doc = "Point module",
    .legacy_methods = PointModuleLegacyMethods,
    .defines = module_defines,
};
```

https://docs.hpyproject.org/en/latest/porting-example/index.html

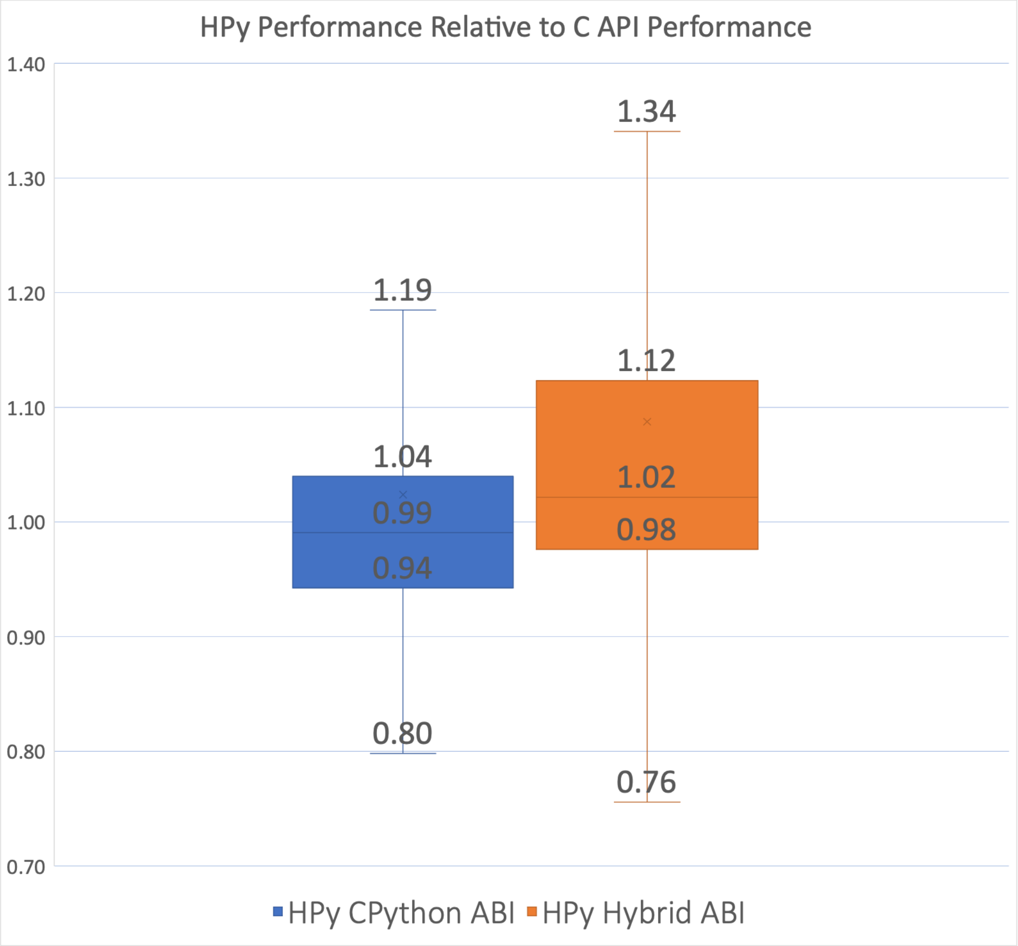
Performance

Kiwi solver port benchmark (8192 inner loops, 10 repetitions)



Performance

Numpy



Debug Mode

- Another suitable name: Strict Mode
- Optional run-time mode (no recompilation required)
- Strictly enforces HPy contract
  - Does additional book keeping for resources
  - Prevents wrong API usage that happens to work by mistake on a certain implementation
- Examples
  - Leaked handles
  - Use-after-close
  - Lifetime of data pointers
  - Read-only memory
  - Invalid reuse of HPy context

Debug Mode: Leak Detector

```
HPyDef_METH(test_leak, "test_leak", HPyFunc_NOARGS)
static HPy
test_leak_stacktrace_impl(HPyContext *ctx, HPy self)
{
    HPy num = HPyLong_FromLong(ctx, 42);
    if (HPy_IsNull(num)) {
        return HPy_NULL;
    }
    // missing HPy_Close(ctx, num);
    return HPy_Dup(ctx, ctx->h_None);
}
```

```
# Run with HPY=debug
import hpy.debug
import snippets

hpy.debug.set_handle_stack_trace_limit(16)
from hpy.debug import LeakDetector

with LeakDetector() as ld:
    snippets.test_leak_stacktrace()
```

```
Traceback (most recent call last):
  File "debug-example.py", line 7, in <module>
    snippets.test_leak_stacktrace()
  File "leakdetector.py", line 43, in __exit__
    self.stop()
  File "leakdetector.py", line 36, in stop
    raise HPyLeakError(leaks)

hpy.debug.leakdetector.HPyLeakError: 1 unclosed handle:
    <DebugHandle 0x556bbcf907c0 for 42>


Allocation stacktrace:

universal.cpython-38d-x86_64-linux-
gnu.so(debug_ctx_Long_FromLong+0x45) [0x7f1d928c48c4]

snippets.hpy.so(+0x122c) [0x7f1d921a622c]

snippets.hpy.so(+0x14b1) [0x7f1d921a64b1]

universal.cpython-38d-x86_64-linux-
gnu.so(debug_ctx_CallRealFunctionFromTrampoline+0xca)
[0x7f1d928bde1e]

snippets.hpy.so(+0x129b) [0x7f1d921a629b]

snippets.hpy.so(+0x1472) [0x7f1d921a6472]

...
```

Universal ABI

DEMO

NumPy/HPy

- A very hard one to migrate
- Why NumPy?
  - If we can do NumPy → we can do everything
- We invested ~1 year (FTE)
- ~180 kLOC (ANSI-C) and 80 kLOC (C API Usage)
  - ~40 kLOC changed
  - ~15 kLOC fully migrated
  - NumPy API: 118 / 261 (45 %)
- We think, hardest part has been done
  - Type migration
  - Metaclass support

https://github.com/hpyproject/numpy-hpy/tree/graal-team/hpy

Current Status

- Current release: 0.9
- We’ve (*partially) migrated several popular packages to HPy
  - ultrajson
  - matplotlib*
  - psutils
  - kiwi solver
  - Pillow*
  - Piconumpy
  - Numpy*
- Cython backend is also planned (PoC exists)

HPy Roadmap (Unordered List)

- Do stable release 0.9.0 (currently: 0.9.0rc2)
- Concentrate on NumPy/HPy
  - Migrate all types to heap types (no HPy involved yet)
  - Evaluate performance
  - Merge upstream
  - Continue migration to HPy
  - Evaluate performance
  - Merge upstream
- Release HPy 1.0.0

Where We Need Help?

- Contributions in form of code, time, or money
- Documentation
  - Core devs concentrate on technical tasks
- Publicity
  - Migrate your package to HPy
  - Write a new package using HPy
- Tooling
  - Anything that helps using HPy
- Packaging
  - Integrate with build systems and PyPI
- Website and logo design

Thank you

ORACLE