PyHEP Numba Tutorial

or

Getting Numba to Do What You Want

Jim Pivarski

Princeton University – IRIS-HEP

February 3, 2021

To optimize or not to optimize?

Scientific computing is of two minds about accelerating code:

> "Tweaking performance should take a back seat to the science."

> "If it doesn't run within time/memory constraints, we can't perform this measurement."

To optimize or not to optimize?

Scientific computing is of two minds about accelerating code:

> "Tweaking performance should take a back seat to the science."

> "If it doesn't run within time/memory constraints, we can't perform this measurement."

Both are true, so it's a balance. The objective: **"time to insight."**

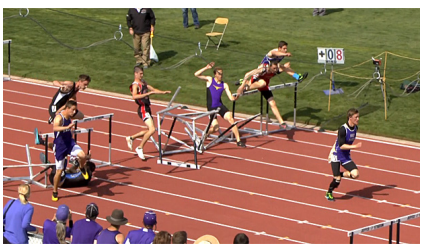
Compilation

Roughly speaking, compilation makes code faster.

Roughly speaking, compilation makes code faster.

More precisely, it removes hurdles to take the slowness out of code.

- Avoids type-checking every value in a loop over identically typed data.
- Avoids following pointers from objects to objects.
- Avoids nonsequential memory access.
- Avoids translating virtual machine instructions into real instructions.
- Avoids garbage collector searches for many small objects.

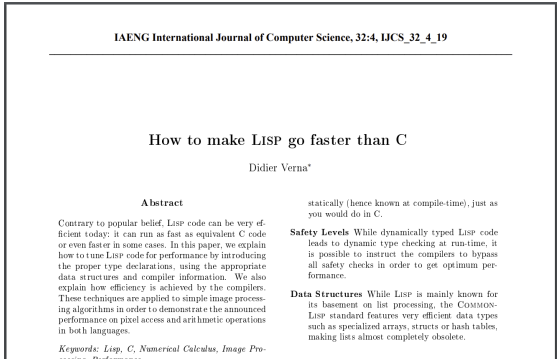


Compilation is not a monolithic thing

Though often thought of as single step applying all optimizations,

```
c++ my_source_code.cpp -o now_its_faster
```

compilation can be piecemeal and applied at any time.



IAENG International Journal of Computer Science, 32:4, IJCS_32_4_19

How to make LISP go faster than C

Didier Verna*

Abstract

Contrary to popular belief, LISP code can be very efficient today: it can run as fast as equivalent C code or even faster in some cases. In this paper, we explain how to tune LISP code for performance by introducing the proper type declarations, using the appropriate data structures and compiler information. We also explain how efficiency is achieved by the compilers. These techniques are applied to simple image processing algorithms in order to demonstrate the announced performance on pixel access and arithmetic operations in both languages.

Keywords: Lisp, C, Numerical Calculus, Image Pro-

statically (hence known at compile-time), just as you would do in C.

Safety Levels While dynamically typed LISP code leads to dynamic type checking at run-time, it is possible to instruct the compilers to bypass all safety checks in order to get optimum performance.

Data Structures While LISP is mainly known for its basement on list processing, the COMMON-LISP standard features very efficient data types such as specialized arrays, structs or hash tables, making lists almost completely obsolete.

Compilers are everywhere

- Python is "compiled" from source code to numeric bytecodes that run in a virtual machine. (At least it isn't scanning the source code over and over!)
- `re.compile("...")` $\longrightarrow$ regular expression to finite state machine.
- `struct.Struct("...")` $\longrightarrow$ same thing for parsing bytes.
- `numexpr.evaluate("...")` $\longrightarrow$ NumExpr's virtual machine for evaluating math formulas is faster than NumPy.
- `ROOT.gInterpreter.Declare("...")` $\longrightarrow$ ROOT's Cling compiles strings of C++ code to machine-native bytecode instructions.
- `rdf.Define("...")` $\longrightarrow$ ROOT's RDataFrame also uses Cling.
- Numba and PyPy compile Python to machine-native bytecode instructions.
- Julia is a language designed around this just-in-time (JIT) compilation.

Compilers are everywhere

- Python is "compiled" from source code to numeric bytecodes that run in a virtual machine. (At least it isn't scanning the source code over and over!)
- `re.compile("...")` ⟶ regular expression to finite state machine.
- `struct.Struct("...")` ⟶ same thing for parsing bytes.
- `numexpr.evaluate("...")` ⟶ NumExpr's virtual machine for evaluating math formulas is faster than NumPy.
- `ROOT.gInterpreter.Declare("...")` ⟶ ROOT's Cling compiles strings of C++ code to machine-native bytecode instructions.
- `rdf.Define("...")` ⟶ ROOT's RDataFrame also uses Cling.
- Numba and PyPy compile Python to machine-native bytecode instructions.
- Julia is a language designed around this **just-in-time (JIT) compilation.**

Just-in-time (JIT) compilers for Python

Runs within normal CPython (can be used with other Python libraries).

Only speeds up functions labeled with the `@numba.jit` decorator.

Only a subset of Python features and functions can be JIT-compiled.

Gains are often factors of 100's.

Replaces the CPython process; not all libraries work/versions of Python exist.

Speeds up the whole program, without any modifications.

All Python features are JIT-compiled.

Gains are often factors of several.

Just-in-time (JIT) compilers for Python

Runs within normal CPython (can be used with other Python libraries).

Only speeds up functions labeled with the `@numba.jit` decorator.

Only a subset of Python features and functions can be JIT-compiled.

Gains are often factors of 100's.

Replaces the CPython process; not all libraries work/versions of Python exist.

Speeds up the whole program, without any modifications.

All Python features are JIT-compiled.

Gains are often factors of several.

What kind of code can Numba accelerate?

- Types must be fully known before the code runs.
- Objects must be replaceable with memory-contiguous values.
- Functions must have or be built out of low-level equivalents.

What kind of code can Numba accelerate?

- Types must be fully known before the code runs.
- Objects must be replaceable with memory-contiguous values.
- Functions must have or be built out of low-level equivalents.

Therefore, only Python libraries that have been specially prepared for Numba work in Numba.

What kind of code can Numba accelerate?

- Types must be fully known before the code runs.
- Objects must be replaceable with memory-contiguous values.
- Functions must have or be built out of low-level equivalents.

Therefore, only Python libraries that have been specially prepared for Numba work in Numba.

Numba optimizes NumPy out-of-the-box, and Awkward Array has been extended as well.

(We're also working on other Scikit-HEP libraries, like Vector and Hist.)

Numba fits into Python's optimization mindset

Most scientific libraries for Python split into a "fast math" part and a "slow bookkeeping" part. Optimization effort must be *focused*.

Numba fits into Python's optimization mindset

Most scientific libraries for Python split into a "fast math" part and a "slow bookkeeping" part. Optimization effort must be *focused*.

Numba isn't about accelerating everything, it's about identifying the part that has to run fast and fixing it.

Numba fits into Python's optimization mindset

Most scientific libraries for Python split into a "fast math" part and a "slow bookkeeping" part. Optimization effort must be *focused*.

Numba isn't about accelerating everything, it's about identifying the part that has to run fast and fixing it.

However, Numba errors can be hard to understand and resolve.

Numba fits into Python's optimization mindset

Most scientific libraries for Python split into a "fast math" part and a "slow bookkeeping" part. Optimization effort must be *focused*.

Numba isn't about accelerating everything, it's about identifying the part that has to run fast and fixing it.

However, Numba errors can be hard to understand and resolve.

That's what this tutorial is about:
how to make Numba happy.