Part I: Introduction to `numba`

Just-in-time Compiled Python for Bioinformatics Research

Johanna Elena Schmitz, Jens Zentgraf and Sven Rahmann

ISMB 2024, Montréal, Canada (July 12, 2024)

Introduction

Why Python is Sometimes Slow

- Python is **interpreted**, not compiled.
  The interpreter reads a Python program and executes it **step by step**.
- Technically, Python is translated to "bytecode" and the bytecode is interpreted.
- Python is **dynamically typed**: A name can represent objects of different types.
- For every operation at run time, the types of the operands need to be determined; then the appropriate code needs to be looked up and executed.

Introduction

Why Python is Sometimes Slow

- Python is **interpreted**, not compiled.
  The interpreter reads a Python program and executes it **step by step**.
- Technically, Python is translated to “bytecode” and the bytecode is interpreted.
- Python is **dynamically typed**: A name can represent objects of different types.
- For every operation at run time, the types of the operands need to be determined; then the appropriate code needs to be looked up and executed.

Solution: Speed up Python with `numba`

- `numba` is a Python package developed by Anaconda, Inc.
- With `numba`, we can **just-in-time compile** (a subset of) Python code to machine code.
- It creates a Python wrapper, so a compiled function can be called from Python.
  This is handled very conveniently by just using a decorator (`@njit`).

First example

```
from numba import njit
@njit
def sum_array(arr):
    return sum(arr)
```

First example

```
from numba import njit
@njit
def sum_array(arr):
    return sum(arr)
```

Benchmarking (for a NumPy array)

- Elapsed time `sum_array` (python mode): 0.038368 seconds.
- Elapsed time `sum_array` with compilation: 0.199138 seconds.
- Elapsed time `sum_array` without compilation: 0.00031 seconds.

First example

```
from numba import njit
@njit
def sum_array(arr):
    return sum(arr)
```

Benchmarking (for a NumPy array)

- Elapsed time `sum_array` (python mode): 0.038368 seconds.
- Elapsed time `sum_array` with compilation: 0.199138 seconds.
- Elapsed time `sum_array` without compilation: 0.00031 seconds.

Explanation

- The first time we call `sum_array` the compilation process is triggered.
- `numba` stores the compiled versions and the original Python version.
- The original Python version can be called with `sum_array.py_func(...)`.

How does `numba` work?

Compilation to Basic CPU Instructions

CPU instructions are simple and **type-specific**.

Addition of two 32-bit integers is different from addition of 64-bit floats.

To compile a Python function directly to machine code,

- the Python code must be “simple” enough
- the CPU instructions must be sufficiently “rich”
- the **types** of all variables must be known (but Python is dynamic)

Approach Taken by `numba`

- The `@njit` decorator replaces the Python function by a `CPUDispatch` object.
- This object **examines the types** of the input arguments
  when the function is **called** (not when it is defined!).
- If no compiled function exists for the given type combination,
  it is compiled (at the time of the first **call**, not when it is defined).
- Python bytecode is translated to “LLVM IR” (intermediate representation).

How Does numba Work?

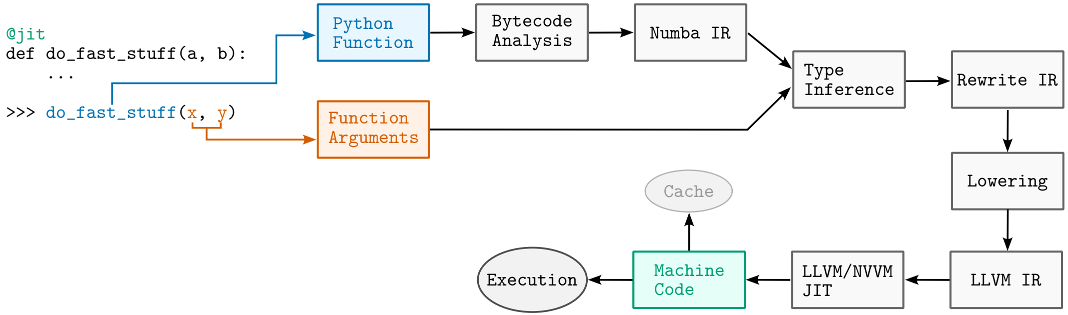

Figure 1: The first time a jit-decorated Python function is called, the above compilation process is triggered.

When Does numba Work Well?

Limitations

1. Most **object oriented** code does not work well with `numba`.
2. Some functions and built-in datatypes don't have a `numba` translation, e.g., `dict`. Parameters and return values are numbers or `numpy` arrays (no objects in general).
3. **File** reading and writing is not easily possible.

When Does `numba` Work Well?

Limitations

1. Most **object oriented** code does not work well with `numba`.
2. Some functions and built-in datatypes don't have a `numba` translation, e.g., `dict`.
   Parameters and return values are numbers or `numpy` arrays (no objects in general).
3. **File** reading and writing is not easily possible.

Possibilities

1. Works very well to solve **numerical problems** (numeric and textual computations).
2. Often achieves significant speed-ups for code with **loops**.
3. Supports most **`numpy`** array operations and functions.
4. Supports simple printing, asserts, exceptions, certain uses of **typed** lists, dicts etc.
5. Supports calling other `njit`-compiled functions.

More Opportunities for Users of `numba`

- We may, from **the same Python code**,
  - compile code specifically for our current CPU,
  - obtain automatic data-level parallelization,
  - obtain semi-automatic thread-level parallelization,
  - compile a GPU kernel for massively data-parallel execution.
- We may extend `numba` to use LLVM primitives from Python. Primitives often directly correspond to machine code (on modern CPUs).
- We may extend `numba` to use functions from the C library (`libc`).
- We may ignore Python's global interpreter lock in compiled functions, and run many **threads** (of compiled functions) in parallel, whereas in Python we can only run multiple (heavy-weight) **processes**.

Late Just-in-Time Compilation

- Compilation takes place during Python program run time anyway.
- A typical easy use case is to decorate top-level functions with `@njit`.
- However, we can also write functions that
  - take parameters
  - `njit`-compile a function based on the given parameters (which are **compile-time constants** at this point)
  - return the resulting compiled function (`CPUDispatch`), with the given parameters "baked in".
- "Late" just-in-time compilation allows for more optimizations.
- Example:
  - `a * b` in general becomes a machine code `mul` instruction.
  - If `b` is a user-specified parameter entered at program start, it can be treated as a constant, e.g. 5.
  - `a * b` becomes `a * 5` or `(a << 2) + a`, which can be faster on some CPUs.

Example

```python
@njit
def multiply_and_power(arr, factor, exponent):
    n = arr.size
    for i in range(n):
        arr[i] = factor * arr[i] ** exponent
```

- Above code will take a `numpy` array `arr`, a scalar factor and exponent, and modify each value of `arr` by taking it to the given power and scaling it.

Example

```
@njit
def multiply_and_power(arr, factor, exponent):
    n = arr.size
    for i in range(n):
        arr[i] = factor * arr[i] ** exponent
```

- Above code will take a numpy array arr, a scalar factor and exponent, and modify each value of arr by taking it to the given power and scaling it.
- We compare the running time of several versions:
  1. pure Python (code as above, but without the @njit decorator)
  2. njit'ted code of the pure Python version (as above)
  3. numpy array operations: arr = factor * arr ** exponent
  4. njit'ted, but with factor and exponent as compile-time constants
  5. like 4., but also using parallel=True
  6. like 4., but compiling a scalar function using @vectorize

Benchmark

Steps

1. We get the `numpy` (imported as `np`) default random number generator.
2. We allocate an array of random integers.
3. We get the current time.
4. We call the function that modifies the array `arr`.
5. We print the elapsed time.
6. We repeat steps 2-5 for all versions.

Benchmark

Timings

Version	Time 1 [s]	Time 2 [s]
Python	12.6698	
Numpy	2.3486	
@njit	1.6304	1.4228
@njit with compile-time constants	0.4969	0.4539
@njit(parallel=True)	0.7909	0.4357
@vectorize	0.9053	0.8684

- Timings may vary with your CPU, machine load, Python version, etc.
- Note that we use a **100x smaller** array for the **pure Python** version, so you should multiply the Python time by 100 to be comparable.
- For `@njit`, the 1st run includes the **compile time** and the 2nd run is without.
- The parallelized version is run with **32 threads**.

Take-Home Messages

- Python alone is not competitive on large data.
- Using `numpy alone` is fine, but has the overhead
  of allocating temporary intermediate arrays for each operation.
  The main bottleneck is often memory.
- Direct `njit` is easy (if your code allows it), and the preferred standard option.
- Using **parameters as compile-time constants** is an optimization
  that is often worth the additional coding effort.
  Always see if it is (easily) possible.
- Parallelization with `parallel=True` and `prange` sometimes helps additionally,
  but can also make it slower. Only use it in rare circumstances.
- Vectorize only applies in certain circumstances (when you need `numpy`
  broadcasting).
  Do not use it in general.

Hands on Session

Download our Git Repository

You can find all the code in the following git repository:

https://gitlab.com/rahmannlab/numba-tutorial

We will later need the T2T genome for motif search. To download the FASTA file run

```
./download_t2t.sh
```

To decompress the file run

```
gzip -dk chm13v2.0.fa.gz
```

Hands on Session

Download our Git Repository

You can find all the code in the following git repository:
https://gitlab.com/rahmannlab/numba-tutorial
We will later need the T2T genome for motif search. To download the FASTA file run
`./download_t2t.sh`
To decompress the file run
`gzip -dk chm13v2.0.fa.gz`

Install conda/mamba Environment

We provide an environment file with all packages we need during this tutorial.
If you have conda/mamba installed, you can create the environment, called **`numbaTutorial`**, using
`mamba env create`
If you don't have a mamba installation, you can download it from
https://github.com/conda-forge/miniforge?tab=readme-ov-file

Try out numba

1. Compare wall clock time of pure python and jit-compiled code.

Run `python multiply_and_power.py` on your laptop and compare the different timings.

2. Write a function `sum_mod_x`.

1. Write a function `sum_mod_x(arr, x)`, that sums all values in an array after taking each value modulo `x`.
2. Now, jit-compile your function using `@njit`.
3. Make `x` a compile-time constant.
4. Compare the timings of the three versions for a `numpy` array of size `1_000_000`.