UNIVERSITÄT DES SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

Part III: Transforming Python 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)

Transforming Python to Numba

1. Identify High- and Low-level Code

- Low level code:
  - Can be translated to LLVM IR
- High level code:
  - Can **not** be translated to LLVM IR
  - File IO
  - Thread management
  - ...

2. Separate Low-level Code

- Move low level code to a function
- Decorate function with `@njit`

3. Data Management

- High level Python objects are not supported
  - set
  - dict
  - ...
- Replace them by NumPy arrays

4. Provide Typing Information

- You can provide typing information for local variables
- `njit(locals=dict(VAR=TYPE)`

Example Running Mean

```
def main(args):
    # High level code
    # Read file containing numbers
    with open(f"array_n{args.n}.csv") as arr_file:
        arr = list(map(int, arr_file.read().split()))

    # Low level code
    asum = sum(arr[:args.window_size])
    out = [asum // args.window_size]
    for i in range(len(arr) - args.window_size):
        asum += arr[i + args.window_size] - arr[i]
        out.append(asum // args.window_size)
```

Example Running Mean

Separating high and low level code

```
def move_mean(array, window_size):
    asum = sum(array[:window_size])
    out = [asum // window_size]
    for i in range(len(array) - window_size):
        asum += array[i + window_size] - array[i]
        out.append(asum // window_size)
    return out

def main(args):
    # Read file containing numbers
    with open(f"array_n{args.n}.csv") as arr_file:
        arr = list(map(int, arr_file.read().split()))
    move_mean(arr, args.window_size)
```

Example Running Mean

Compiling low level code

```
@njit
def move_mean(array, window_size):
    asum = sum(array[:window_size])
    out = [asum // window_size]
    for i in range(len(array) - window_size):
        asum += array[i + window_size] - array[i]
        out.append(asum // window_size)
    return out

def main(args):
    # Read file containing numbers
    with open(f"array_n{args.n}.csv") as arr_file:
        arr = list(map(int, arr_file.read().split()))
    move_mean(arr, args.window_size)
```

Data Management

Supported data types

- int
- float
- bool
- str
- NumPy arrays and types
- Classes (early version)
- Typed dicts
- Typed lists

Unsupported data types

- Heterogeneous set
- Heterogeneous Python list
- Heterogeneous tuple (limited)
- Python dict

Example Running Mean

Replacing Lists with NumPy arrays

```
@njit
def move_mean(array, window_size, out):
    asum = sum(array[:window_size])
    out[0] = asum // window_size
    for i in range(len(array) - window_size):
        asum += array[i + window_size] - array[i]
        out[i + 1] = asum // window_size

def main(args):
    # Read file containing numbers
    with open(f"array_n{args.n}.csv") as arr_file:
        arr = np.array(list(map(int, arr_file.read().split())),
                       dtype=np.int64)
    out = np.empty(len(arr) - args.window_size + 1, dtype=np.int64)
    move_mean(arr, args.window_size, out)
```

Type Annotations

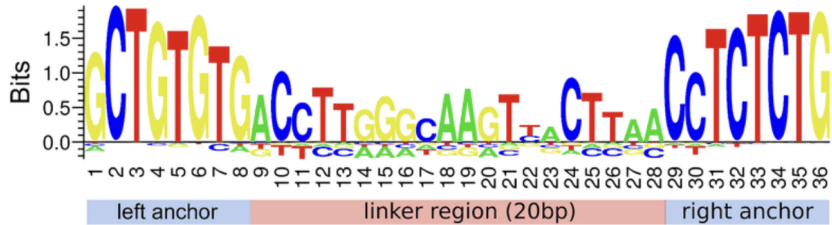
- Type information are used to lower the Python binary code to LLVM IR.
- Can be determined by Numba during the run time.
- Can be directly specified.
- Setting the types can provide a speed up.
- Specify types: `@njit(locals=dict(NAME=VALUE))`.

Example Running Mean

```
@njit(locals=dict(asum=int64))
def move_mean(array, window_size, out):
    asum = sum(array[:window_size])
    out[0] = asum // window_size
    for i in range(len(array) - window_size):
        asum += array[i + window_size] - array[i]
        out[i + 1] = asum // window_size

def main(args):
    # Read file containing numbers
    with open(f"array_n{args.n}.csv") as arr_file:
        arr = np.array(list(map(int, arr_file.read().split())),
                       dtype=np.int64)
    out = np.empty(len(arr) - args.window_size + 1, dtype=np.int64)
    move_mean(arr, args.window_size, out)
```

Transforming Motif Matcher



Steps

1. Identify high- and low-level code.
2. Separate low-level code.
3. Adjust data management.
4. Provide type information.

Transforming Motif Matcher

- High and low level code is already separated.
- Generators are not fully supported.
  $\rightarrow$ Replace with a function.
- Create an output NumPy array to store the results.
  - How many hits do we have?
  - Differs for each chromosome.
  - Set a fixed upper limit.

```
# Pure Python implementation
def find_matches(mask, I, F, accept, sequence, *_):
    """
    generator yielding all end positions
    """
    A = 0
    for i, c in enumerate(sequence):
        A = ((A << 1) | 1) & int(mask[c])
        A = A | ((F - (A & I)) & ~F)
        if A & accept:
            yield i

for header, sequence in fasta_items(args.fasta):
    print("#", header.decode("ASCII"))
    for pos in find_matches_slow(*nfa, sequence):
        print(pos)
```

Transforming Motif Matcher

1. Restructure the generator function.
2. Store matches positions in an NumPy array.

```
def find_matches(mask, I, F, accept, sequence, out):
    k = 0  # number of found positions
    N = results.size
    A = 0
    for i, c in enumerate(sequence):
        A = ((A << 1) | 1) & int(mask[c])  # int vs. numpy.uint64
        A = A | ((F - (A & I)) & ~F)
        if A & accept:
            # store position in output
            if k < N: results[k] = i
            k += 1  # increase the number of found positions
    return k
```

Transforming Motif Matcher

1. Create the output array.
2. Adapt to the new function.

```
for header, sequence in fasta_items(args.fasta):
    print("#", header.decode("ASCII"))
    # Create NumPy array to store the results
    results = np.empty(NRESULTS, dtype=np.uint32)
    nresults = find_matches(*nfa, sequence, results):
    if nresults > NRESULTS:
        print(f"! Too many results, showing first {NRESULTS}")
        nresults = NRESULTS
    print(*list(results[:nresults]), sep="\n")
```

Transforming Motif Matcher

1. Add `@njit` decorator.
2. Specify types of local variables.

```
@njit(locals=dict(k=uint64, N=uint64, A=uint64))
def find_matches(mask, I, F, accept, sequence, out):
    k = 0
    N = results.size
    A = 0
    for i, c in enumerate(sequence):
        A = ((A << 1) | 1) & int(mask[c])
        A = A | ((F - (A & I)) & ~F)
        if A & accept:
            if k < N: results[k] = i
            k += 1
    return k
```

Take-Home Messages

4 easy steps:

1. Separate high and low level code.
2. Compile low level code using `@njit`.
3. Define types of local variables.
4. Adapt high level code.

- If the code is already structured, it is often enough to just add `@njit`.
- Use NumPy arrays instead of lists or generators.