

Modelling, Uncertainty and Data for Engineers (MUDE)

Vectorization in Python and Numba

Denis Voskov, Ilshat Saifullin

Introduction

In this session we will learn how to make your code more efficient using

- Numpy vectorization - operations delegate the looping internally to highly optimized functions
- Numba which analyzes your function and compiles it into fast machine code using the LLVM (Low Level Virtual Machine) compiler
- Parallel Numba which automatically parallelize loops using multi-core architecture of modern CPU

Problem

Generate a random integer array of ages with interval 1-100. Calculate the average adult age.
Measure the calculation time.

```
from numpy import random

# generate random array of integers from 0 to 100 with size 1e+6.
ages = random.randint(0, 100, 1000000)

# calculate the average age
average_age = ages.mean()
print("Average age = ", average_age)
```

Solution using the loop

```
# calculate the average age only for an adults
(age >= 18) using loop
def calc_average_adult_age_loop(ages):
    average_adult_age = 0
    adult_counter = 0
    for i in ages:
        if i >= 18:
            average_adult_age += i
            adult_counter += 1
    if (adult_counter > 0):
        average_adult_age /= adult_counter
    return average_adult_age
```

Solution using numpy

```
# calculate and measure the calculation time
import time
start = time.time()
average_adult_age = calc_average_adult_age_loop(ages)
end = time.time()
print("Time (loop solution)", end - start)
print("Average adult age = ", average_adult_age)
```

Wall time: 180 ms

```
# calculate the average age only for an adults (age >= 18) using numpy's vectorization
start = time.time()
average_adult_age = ages[ages>=18].mean()
end = time.time()
print("Time (vectorization solution)", end - start)
print("Average adult age = ", average_adult_age)
```

Wall time: 5 ms

Numba

Numba speeds up your Python functions by translating them to optimized machine code

```
from numba import jit

@jit(nopython=True)
def calc_average_adult_age_loop(ages):
    average_adult_age = 0
    adult_counter = 0
    for i in ages:
        if i >= 18:
            average_adult_age += i
            adult_counter += 1
    if (adult_counter > 0):
        average_adult_age /= adult_counter
    return average_adult_age
```

Numba

Measuring the calculation time with numba

```
# first call will include the compilation time
```

```
start = time.time()
average_adult_age = calc_average_adult_age_loop(ages)
end = time.time()
print("Time (vectorization solution)", end - start)
print("Average adult age = ", average_adult_age)
```

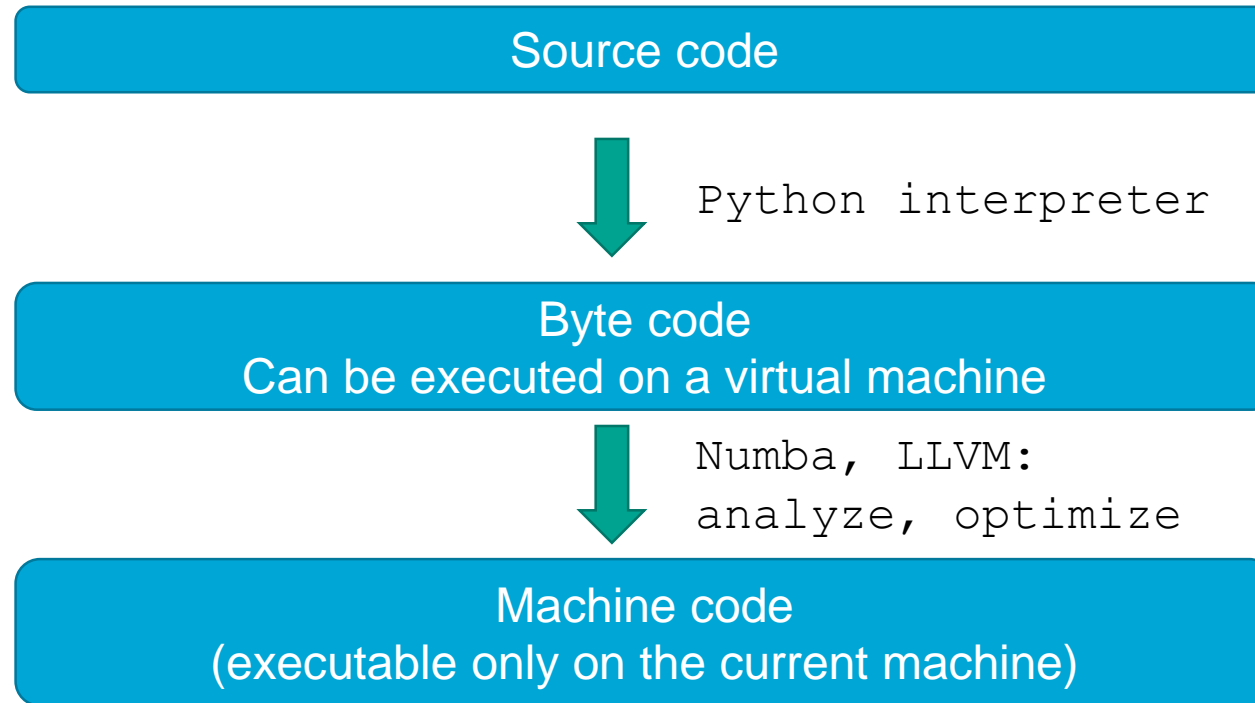
Wall time: 321 ms

```
# second call should be faster
```

```
start = time.time()
average_adult_age = calc_average_adult_age_loop(ages)
end = time.time()
print("Time (vectorization solution)", end - start)
print("Average adult age = ", average_adult_age)
```

Wall time: 1 ms

How Numba works



Parallelization

Threading allows different parts of your program to run concurrently on a single computer (with shared memory)

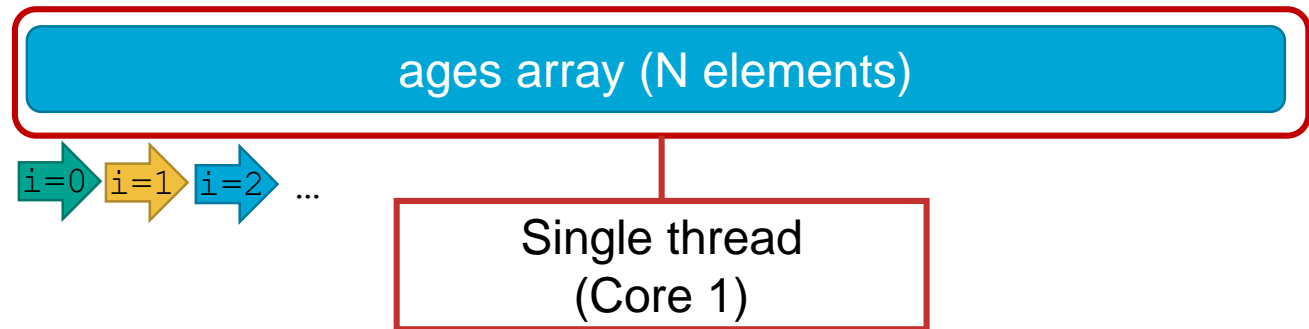
To make the code parallel:

- Identify independent tasks
- Chunk the work
- Redesign the code if need

Parallelization (on example with computing mean value)

```
sum_ages = 0.0
for i in range(N):
    sum_ages += ages[i]
mean = sum_ages / N
```

Serial: executing iterations one by one



Other CPU cores do nothing

Parallelization (on example with computing mean value)

Most of work is in the loop.

1. No dependency on previous iterations.
2. And summation allows permutation.

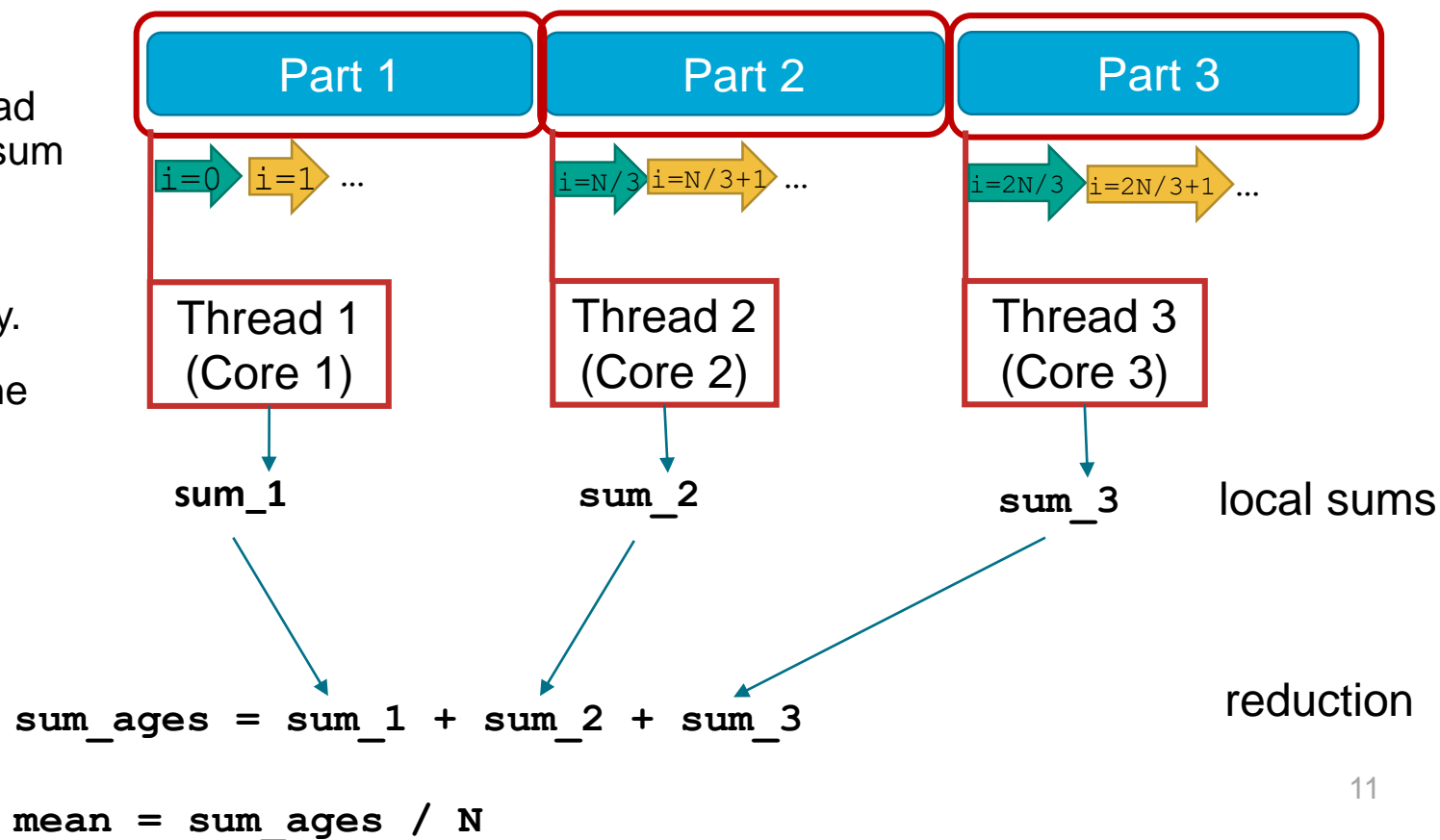
So it is possible to split the loop. Each thread processes its part and computes the local sum **sum_i**.

We cannot use the same variable for sum, because threads write into it simultaneously.

Afterwards, one of the threads computes the global sum.

Computation of mean is only one division operation, and it is done sequentially.

Parallel: Executing few iterations at the same time



Parallelization (on example with computing mean value)

```
mean = 0.0
@jit(nopython=True, parallel=True)
for i in prange(N):
    mean += ages[i]
mean /= N
```

We ask numba to:

- create threads and split the loop between them
- analyse code inside the loop and manage to create additional temporary variables to safely compute local sums
- execute the loop in parallel using available CPU cores
- synchronize threads at the end of the loop
- compute the global sum from local sums