LESSON NAME

Intro

> Prerequisites
>
> - FIXME
> - ...
> - ...

20 min	filename

Cython

Cython is a superset of Python that additionally supports calling C functions and declaring C types on variables and class attributes. It is also a versatile, general purpose compiler. Since it is supports a superset of Python syntax, nearly all Python code, including 3rd party Python packages are also valid Cython code. Under Cython, source code gets translated into optimized C/C++ code and compiled as Python extension modules.

Developers can either:

- prototype and develop Python code in IPython/Jupyter using the `%%cython` magic command (**easy**), or
- run the `cython` command-line utility to produce a `.c` file from a `.py` or `.pyx` file, which in turn needs to be compiled with a C compiler to an `.so` library, which can then be directly imported in a Python program (**intermediate**), or
- use setuptools or meson with meson-python to automate the aforementioned build process (**advanced**).

Herein, we restrict the discussion to the Jupyter-way of using the `%%cython` magic. A full overview of Cython capabilities refers to the documentation.

> Important
>
> Due to a known issue with `%%cython -a` in `jupyter-lab` we have to use the `jupyter-nbclassic` interface for this

Skip to content

Python: Baseline (step 0)

> Demo: Cython
>
> Consider a problem to integrate a function:
>
> $$I = \int^{b}_{a}(x^2 - x)dx$$
>
> which can be numerically approximated as the following sum:
>
> $$I \approx \delta x \sum_{i=0}^{N-1} (x_i^2 - x_i)$$
>
> where $a \le x_i \lt b$, and all x_i are uniformly spaced apart by $\delta x = (b - a) / N$.
>
> **Objective**: Repeatedly compute the approximate integral for 1000 different combinations of a, b and N.

Python code is provided below:

```
import numpy as np

def f(x):
    return x ** 2 - x

def integrate_f(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx

def apply_integrate_f(col_a, col_b, col_N):
    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f(col_a[i], col_b[i], col_N[i])
    return res
```

We generate a dataframe and apply the `apply_integrate_f()` function on its columns, timing the execution:

```
import pandas as pd

df = pd.DataFrame(
    {
        "a": np.random.randn(1000),
        "b": np.random.randn(1000),
        "N": np.random.randint(low=100, high=1000, size=1000)
    }
)

%timeit apply_integrate_f(df['a'], df['b'], df['N'])
# 101 ms ± 736 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Cython: Benchmarking (step 1)

In order to use Cython, we need to import the Cython extension:

```
%load_ext cython
```

As a first cythonization step, we add the cython magic command (`%%cython -a`) on top of Jupyter code cell. We start by a simply compiling the Python code using Cython without any changes. The code is shown below:

```
%%cython -a

import numpy as np

def f_cython_step1(x):
    return x * (x - 1)

def integrate_f_cython_step1(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_cython_step1(a + i * dx)
    return s * dx

def apply_integrate_f_cython_step1(col_a, col_b, col_N):
    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_cython_step1(col_a[i], col_b[i], col_N[i])
    return res
```

Skip to content

Our task is to remove as much yellow as possible by *static typing*, *i.e.* explicitly declaring arguments, parameters, variables and functions.

Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

```
 01:
+02: import numpy as np
 03:
+04: def f_cython_step1(x):
+05:     return x * (x - 1)
 06:
+07: def integrate_f_cython_step1(a, b, N):
+08:     s = 0
+09:     dx = (b - a) / N
+10:     for i in range(N):
+11:         s += f_cython_step1(a + i * dx)
+12:     return s * dx
 13:
+14: def apply_integrate_f_cython_step1(col_a, col_b, col_N):
+15:     n = len(col_N)
+16:     res = np.empty(n,dtype=np.float64)
+17:     for i in range(n):
+18:         res[i] = integrate_f_cython_step1(col_a[i], col_b[i], col_N[i])
+19:     return res
```

ANNOTATED CYTHON CODE OBTAINED BY RUNNING THE CODE ABOVE. THE YELLOW COLORING IN THE OUTPUT SHOWS US THE AMOUNT OF PURE PYTHON CODE.

We benchmark the Python code just using Cython, and it may give either similar or a slight increase in performance.

```
%timeit apply_integrate_f_cython_step1(df['a'], df['b'], df['N'])
# 102 ms ± 2.06 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Cython: Adding data type annotation to input variables (step 2)

Now we can start adding data type annotation to the input variables as highlightbed in the code example/cython below:

Pure Python | Cython

```
%%cython -a

import cython
import numpy as np


def f_cython_step2(x: cython.double):
    return x ** 2 - x

def integrate_f_cython_step2(a: cython.double, b: cython.double, N: cython.long):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_cython_step2(a + i * dx)
    return s * dx

def apply_integrate_f_cython_step2(
    col_a: cython.double[:],
    col_b: cython.double[:],
    col_N: cython.long[:],
):
    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_cython_step2(col_a[i], col_b[i], col_N[i])
    return res
```

```
# this will not work
#%timeit apply_integrate_f_cython_step2(df['a'], df['b'], df['N'])

# this command works (see the description below)
%timeit apply_integrate_f_cython_step2(df['a'].to_numpy(), df['b'].to_numpy(), df['N']
# 34.3 ms ± 537 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Warning

You can not pass a Series directly since Cython definition is specific to an array. Instead we should use `Series.to_numpy()` to get the underlying NumPy array which works nicely with Cython.

Note

Cython uses the normal C syntax for types and provides all standard ones, including pointers. Here is a list of some primitive C data types (refer to Cython's documentation on Types):

Cython type identifier	Pure Python dtype
`char`	`cython.char`
`int`	`cython.int`
`unsigned int`	`cython.uint`
`long`	`cython.long`
`float`	`cython.float`
`double`	`cython.double`
`double complex`	`cython.doublecomplex`
`size_t`	`cython.size_t`

Using these data types, we can also annotate arrays (see Typed Memoryviews):

- 1D `np.float64` array would be equivalent to `cython.double[:]`,
- 2D `np.float64` array would be equivalent to `cython.double[:, :]` and so on…

Important

to quote the Cython documentation,

> **Typing is not a necessity**
>
> Providing static typing to parameters and variables is convenience to speed up your code, but it is not a necessity. Optimize where and when needed. In fact, typing can slow down your code in the case where the typing does not allow optimizations but where Cython still needs to check that the type of some object matches the declared type.

Cython: Adding data type annotation to functions (step 3)

Next step, we further add type annotation to functions. There are three ways of declaring functions:

- `def` - Python style:
  - Called by Python or Cython code, and both input/output are Python objects.
  - Declaring argument types and local types (thus return values) can allow Cython to generate optimized code which speeds up the execution.
  - Once types are declared, a `TypeError` will be raised if the function is passed with the wrong types.
- `@cython.cfunc` or `cdef` - C style:
  - cdef functions are called from Cython and C, but not from Python code.
  - Cython treats functions as pure C functions, which can take any type of arguments, including non-Python types, *e.g.*, pointers.
  - This usually gives the *best performance*.
  - However, one should really take care of the functions declared by `cdef` as these functions are actually writing in C.
- `@cython.ccall` or `cpdef` - C/Python mixed style:
  - cpdef function combines both `cdef` and `def`.
  - Cython will generate a `cdef` function for C types and a `def` function for Python types.
  - In terms of performance, `cpdef` functions may be *as fast as* those using `cdef` and might be as slow as `def` declared functions.

Pure Python | Cython

```
%%cython -a

import cython
import numpy as np


@cython.cfunc
def f_cython_step3(x: cython.double):
    return x ** 2 - x

@cython.cfunc
def integrate_f_cython_step3(a: float, b: float, N: int):
    s = 0
    dx = (b - a) / N

    for i in range(N):
        s += f_cython_step3(a + i * dx)
    return s * dx

@cython.ccall
def apply_integrate_f_cython_step3(
    col_a: cython.double[:],
    col_b: cython.double[:],
    col_N: cython.long[:]
):
    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_cython_step3(col_a[i], col_b[i], col_N[i])
    return res
```

```
%timeit apply_integrate_f_cython_step3(df['a'].to_numpy(), df['b'].to_numpy(), df['N']
# 29.2 ms ± 152 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Pure Python | Cython

```
%%cython -a

import cython
import numpy as np


def f_cython_step2(x: cython.double):
    return x ** 2 - x

def integrate_f_cython_step2(a: cython.double, b: cython.double, N: cython.long):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_cython_step2(a + i * dx)
    return s * dx

def apply_integrate_f_cython_step2(
    col_a: cython.double[:],
    col_b: cython.double[:],
    col_N: cython.long[:],
):
    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_cython_step2(col_a[i], col_b[i], col_N[i])
    return res
```

Cython: Adding data type annotation to local variables and return (step 4)

Last step, we can add type annotation to local variables within functions and the return value.

Pure Python | Cython

```
%%cython -a

import cython
import numpy as np

@cython.cfunc
def f_cython_step4(x: cython.double) -> cython.double:
    return x ** 2 - x

@cython.cfunc
def integrate_f_cython_step4(
    a: cython.double,
    b: cython.double,
    N: cython.long
) -> cython.double:
    s: cython.double
    dx: cython.double
    i: cython.long

    s = 0
    dx = (b - a) / N

    for i in range(N):
        s += f_cython_step4(a + i * dx)
    return s * dx

@cython.ccall
def apply_integrate_f_cython_step4(
    col_a: cython.double[:],
    col_b: cython.double[:],
    col_N: cython.long[:]
) -> cython.double[:]:
    n: cython.int
    i: cython.int
    res: cython.double[:]

    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_cython_step4(col_a[i], col_b[i], col_N[i])
    return res
```



```
_integrate_f_cython_step4(df['a'].to_numpy(), df['b'].to_numpy(), df['N']
# 471 µs ± 7.38 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Now it is ~200 times faster than the baseline Python implementation, and all we have done is to add type declarations on the Python code!

Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

```
 01:
+02: import cython
+03: import numpy as np
 04:
+05: @cython.cfunc
 06: def f_cython_step4(x: cython.double) -> cython.double:
+07:     return x ** 2 - x
 08:
+09: @cython.cfunc
 10: def integrate_f_cython_step4(
 11:     a: cython.double,
 12:     b: cython.double,
 13:     N: cython.long
 14: ) -> cython.double:
 15:     s: cython.double
 16:     dx: cython.double
 17:     i: cython.long
 18:
+19:     s = 0
+20:     dx = (b - a) / N
 21:
+22:     for i in range(N):
+23:         s += f_cython_step4(a + i * dx)
+24:     return s * dx
 25:
+26: @cython.ccall
 27: def apply_integrate_f_cython_step4(
 28:     col_a: cython.double[:],
 29:     col_b: cython.double[:],
 30:     col_N: cython.long[:]
 31: ) -> cython.double[:]:
 32:     n: cython.int
 33:     i: cython.int
 34:     res: cython.double[:]
 35:
+36:     n = len(col_N)
+37:     res = np.empty(n,dtype=np.float64)
+38:     for i in range(n):
+39:         res[i] = integrate_f_cython_step4(col_a[i], col_b[i], col_N[i])
+40:     return res
```

WE INDEED SEE MUCH LESS PYTHON INTERACTION IN THE CODE FROM STEP 1 TO STEP 4.

Other useful features

There are some useful (and possibly advanced) features which are not covered in this episode. Some of these features are called magic attributes. Here are a few:

- `cython.cimports` package for importing and calling C libraries such as libc.math.

Note

Differences between `import` (for Python) and `cimport` (for Cython) statements

- `import` gives access to Python libraries, functions or attributes
- `cimport` gives access to C libraries, functions or attributes

In case of Numpy it is common to use the following, and Cython will internally handle this ambiguity.

Pure Python | Cython

```
from cython.cimports.libc.stdlib import malloc, free  # Allocate and free memory
from cython.cimports.libc import math  # For math functions like sin, cos etc.
from cython.cimports import numpy as np # access to NumPy C API
```

- `cython.nogil`, which can act both as a decorator or context-manager, to manage the GIL (Global Interpreter Lock). See Cython and the GIL.
- `@cython.boundscheck(False)` and `@cython.wraparound(False)` decorators to tune indexing of Numpy array. See Cython for NumPy users.
- `@cython.cclass` to declare Extension Types which behave similar to Python classes.

In addition to the above Cython can also,

- augment with .pxd files where the Python code is kept as it is and the `.pxd` file describes the type annotation. In this form `.pxd` is very similar in function to a C/C++ header file or `.pyi` Python type annnotation file,
- create parallel code using parallel blocks and `prange` iterator for element-wise parallel operation or reductions based on OpenMP threads (see Writing parallel code with Cython).

Demo

Here is a code which showcases most of the features above, except the `@cython.cclass` feature and the use of `.pxd` files.

Pure Python | Cython | Numpy | Naive Python implementation

```
import cython
from cython.parallel import parallel, prange
from cython.cimports.libc.math import sqrt

@cython.boundscheck(False)
@cython.wraparound(False)
def normalize(x: cython.double[:]):
    """Normalize a 1D array by dividing all its elements using its root-mean-square (RMS) value."""
    i: cython.Py_ssize_t
    total: cython.double = 0
    norm: cython.double
    with cython.nogil, parallel():
        for i in prange(x.shape[0]):
            total += x[i]*x[i]
        norm = sqrt(total)
        for i in prange(x.shape[0]):
            x[i] /= norm
```

Note

If you compare performance of the the Cython code versus the Numpy code, you might observe that it is either on-par, or slightly worse than Numpy. This is because Numpy vectorized operations also makes use of OpenMP parallelism and is heavily optimized. Nevertheless, it is orders of magnitude better than a naive implementation.

Conclusions

Keypoints

- Cython is a versatile, general purpose compiler for Python code
- Cython is a great way to write high-performance code in Python where algorithms are not available in scientific libraries like Numpy and Scipy and require custom implementation

See also

In order to make Cython code reusable often some packaging is necessary. The compilation to binary extension can either happen during the packaging itself, or during installation of a Python package. To learn more about how to

Skip to content

sions, read the following guides:

- *pyOpenSci Python packaging guide*'s page on build tools
- *Python packaging user guide*'s page on packaging binary extensions

CuPy

Questions

- How could I make my Python code to run on a GPU?
- How do I copy data to the GPU memory?

Objectives

- Understand the basics of the library CuPy and its functionalities
- Analyze and detect whether a variable is stored in the CPU or GPU memory
- Execute a data-copy operation from host to device memory and vice versa
- Re-write a simple NumPy/SciPy function, to program the CuPy equivalent which runs on the GPUs

Introduction to CuPy

Another excellent tool for writing Python code to run on GPUs is CuPy. CuPy implements most of the NumPy/SciPy operations and acts as a drop-in replacement to run existing code on both NVIDIA CUDA or AMD ROCm platforms. By design, the CuPy interface is as close as possible to NumPy/SciPy, making code porting much easier.

Note

A common misconception is that CuPy is an official NVIDIA project. It is rather a community driven project. Originally it was developed to support a deep-learning framework called Chainer (now deprecated), wherein it only supported CUDA as a target. Nowadays CuPy has great support for both NVIDIA CUDA or AMD ROCm platforms.

Basics of CuPy

CuPy's syntax here is identical to that of NumPy. A list of NumPy/SciPy APIs and its corresponding CuPy implementations is summarised here:

Complete Comparison of NumPy and SciPy to CuPy functions.

In short, CuPy provides N-dimensional array (ndarray), sparse matrices, and the associated routines for GPU devices, most having the same API as NumPy/SciPy.

Let us take a look at the following code snippet which calculates the L2-norm of an array. Note how simple it is to run on a GPU device using CuPy, i.e. essentially by changing np to cp.

NumPy

```
import numpy as np
x_cpu = np.array([1, 2, 3])
l2_cpu = np.linalg.norm(x_cpu)
```

CuPy

```
import cupy as cp
x_gpu = cp.array([1 ,2 ,3])
l2_gpu = cp.linalg.norm(x_gpu)
```

> **Warning**
>
> Do not change the import line in the code to something like
>
> `import cupy as np`
>
> which can cause problems if you need to use NumPy code and not CuPy code.

Conversion to/from NumPy arrays

Although `cupy.ndarray` is the CuPy counterpart of NumPy `numpy.ndarray`, the main difference is that `cupy.ndarray` resides on the `current device`, and they are not implicitly convertible to each other. When you need to manipulate CPU and GPU arrays, an explicit data transfer may be required to move them to the same location – either CPU or GPU. For this purpose, CuPy implements the following methods:

- To convert `numpy.ndarray` to `cupy.ndarray`, use `cupy.array()` or `cupy.asarray()`
- To convert `cupy.ndarray` to `numpy.ndarray`, use `cupy.asnumpy()` or `cupy.ndarray.get()`

These methods can accept arbitrary input, meaning that they can be applied to any data that is located on either the host or device.

Here is an example that demonstrates the use of both methods:

```
>>> import numpy as np
>>> import cupy as cp
>>>
>>> x_cpu = np.array([1, 2, 3]) # allocating array x on cpu
>>> y_cpu = np.array([4, 5, 6]) # allocating array y on cpu
>>> x_cpu + y_cpu # add x and y
array([5, 7, 9])
>>>
>>> x_gpu = cp.asarray(x_cpu) # move x to gpu
>>> x_gpu + y_cpu # now it should fail
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "cupy/_core/core.pyx", line 1375, in cupy._core.core._ndarray_base.__add__
  File "cupy/_core/core.pyx", line 1799, in cupy._core.core._ndarray_base.__array_ufunc
  File "cupy/_core/_kernel.pyx", line 1285, in cupy._core._kernel.ufunc.__call__
  File "cupy/_core/_kernel.pyx", line 159, in cupy._core._kernel._preprocess_args
  File "cupy/_core/_kernel.pyx", line 145, in cupy._core._kernel._preprocess_arg
TypeError: Unsupported type <class 'numpy.ndarray'>
>>>
>>> cp.asnumpy(x_gpu) + y_cpu
array([5, 7, 9])
>>> cp.asnumpy(x_gpu) + cp.asnumpy(y_cpu)
array([5, 7, 9])
>>> x_gpu + cp.asarray(y_cpu)
array([5, 7, 9])
>>> cp.asarray(x_gpu) + cp.asarray(y_cpu)
array([5, 7, 9])
```

> **Note**
>
> Converting between `cupy.ndarray` and `numpy.ndarray` incurs data transfer between the host (CPU) device and the GPU device, which is costly in terms of performance.

Current Device

CuPy introduces the concept of a `current device`, which represents the default GPU device on which the allocation, manipulation, calculation, etc., of arrays take place. `cupy.ndarray.device` attribute can be used to determine the device allocated to a CuPy array. By default, ID of the current device is 0.

```
>>> import cupy as cp
>>> x_gpu = cp.array([1, 2, 3, 4, 5])
>>> x_gpu.device
```


To obtain the total number of accessible devices, one can utilize the `getDeviceCount` function:

```
>>> import cupy as cp
>>> cp.cuda.runtime.getDeviceCount()
1
```

To switch to another GPU device, use the `Device` context manager. For example, the following code snippet creates an array on GPU 1:

```
>>> import cupy as cp
>>> with cp.cuda.Device(1):
       x_gpu1 = cp.array([1, 2, 3, 4, 5])
>>> print("x_gpu1 is on device:" x_gpu1.device)
```

Sometimes it is more convenient to set the device globally:

```
>>> import cupy as cp
>>> cp.cuda.runtime.setDevice(1)
```

All CuPy operations (except for multi-GPU features and device-to-device copy) are performed on the currently active device.

> Note
>
> The device will be called <CUDA Device 0> even if you are on AMD GPUs.
>
> In general, CuPy functions expect that the data array is on the current device. Passing an array stored on a non-current device may work depending on the hardware configuration but is generally discouraged as it may not be performant.

Exercises: Matrix Multiplication

> Exercise : Matrix Multiplication
>
> The first example is a simple matrix multiplication in single precision (float32). The arrays are created with random values in the range of -1.0 to 1.0. Convert the NumPy code to run on GPU using CuPy.

```
import math
import numpy as np

A = np.random.uniform(low=-1., high=1., size=(64,64)).astype(np.float32)
B = np.random.uniform(low=-1., high=1., size=(64,64)).astype(np.float32)
C = np.matmul(A,B)
```

Solution

```
import math
import cupy as cp

A = cp.random.uniform(low=-1., high=1., size=(64,64)).astype(cp.float32)
B = cp.random.uniform(low=-1., high=1., size=(64,64)).astype(cp.float32)
C = cp.matmul(A,B)
```

Notice in this snippet of code that the variable C remains on the GPU. You have to copy it back to the CPU explicitly if needed. Otherwise all the data on the GPU is wiped once the code ends.

Exercises: moving data from GPU to CPU

Exercise : moving data from GPU to CPU

The code snippet simply computes a singular value decomposition (SVD) of a matrix. In this case, the matrix is a single-precision 64x64 matrix of random values. First re-write the code using CuPy for GPU enabling. Second, adding a few lines to copy variable u back to CPU and print objects' type using `type()` function.

```
import numpy as np

A = np.random.uniform(low=-1., high=1., size=(64, 64)).astype(np.float32)
u, s, v = np.linalg.svd(A)
```

Solution

```
import cupy as cp

A = cp.random.uniform(low=-1., high=1., size=(64, 64)).astype(cp.float32)
u_gpu, s_gpu, v_gpu = cp.linalg.svd(A)
print("type(u_gpu) = ",type(u_gpu))
u_cpu = cp.asnumpy(u_gpu)
print("type(u_cpu) = ",type(u_cpu))
```

Exercises: CuPy vs Numpy/SciPy

CuPy vs Numpy/SciPy

Although the CuPy team focuses on providing a complete NumPy/SciPy API coverage to become a full drop-in replacement, some important differences between CuPy and NumPy should be noted, one should keep these differences in mind when porting NumPy code to CuPy.

Here are various examples illustrating the differences

When CuPy is different from NumPy/SciPy

Cast behavior from float to integer

Some casting behaviors from float to integer are not defined in C++ specification. The casting from a negative float to unsigned integer and infinity to integer is one of such examples. The behavior of NumPy depends on your CPU architecture. This is the result on an Intel CPU:

```
>>> np.array([-1], dtype=np.float32).astype(np.uint32)
array([4294967295], dtype=uint32)

>>> cp.array([-1], dtype=np.float32).astype(np.uint32)
array([0], dtype=uint32)
```

```
>>> np.array([float('inf')], dtype=np.float32).astype(np.int32)
array([-2147483648], dtype=int32)

>>> cp.array([float('inf')], dtype=np.float32).astype(np.int32)
array([2147483647], dtype=int32)
```

Random methods support dtype argument

NumPy's random value generator does not support a dtype argument and instead always returns a float64 value. While in CuPy, both float32 and float64 are supported because of cuRAND.

```
>>> np.random.randn(dtype=np.float32)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: randn() got an unexpected keyword argument 'dtype'

>>> cp.random.randn(dtype=np.float32)
array(1.3591791, dtype=float32)
```

Out-of-bounds indices

CuPy handles out-of-bounds indices differently by default from NumPy when using integer array indexing. NumPy handles them by raising an error, but CuPy wraps around them.

```
>>> x = np.array([0, 1, 2])
>>> x[[1, 3]] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: index 3 is out of bounds for axis 0 with size 3

>>> x = cp.array([0, 1, 2])
>>> x[[1, 3]]
array([1, 0])
>>> x[[1, 3]] = 10
>>> x
array([10, 10,  2])
```


in indices

CuPy's `__setitem__` behaves differently from NumPy when integer arrays reference the same location multiple times. NumPy stores the value corresponding to the last element among elements referencing duplicate locations. In CuPy, the value that is actually stored is undefined.

```
>>> a = np.zeros((2,))
>>> i = np.arange(10000) % 2
>>> v = np.arange(10000).astype(np.float32)
>>> a[i] = v
>>> a
array([9998., 9999.])

>>> a = cp.zeros((2,))
>>> i = cp.arange(10000) % 2
>>> v = cp.arange(10000).astype(cp.float32)
>>> a[i] = v
>>> a
array([4592., 4593.])
```

Zero-dimensional array

Reduction methods

NumPy's reduction functions (e.g. `numpy.sum()`) return scalar values (e.g. `numpy.float32`). However CuPy counterparts return zero-dimensional `cupy.ndarray`. That is because CuPy scalar values (e.g. `cupy.float32`) are aliases of NumPy scalar values and are allocated in CPU memory. If these types were returned, it would be required to synchronize between GPU and CPU. If you want to use scalar values, cast the returned arrays explicitly.

```
>>> type(np.sum(np.arange(3))) == np.int64
True

>>> type(cp.sum(cp.arange(3))) == cp.ndarray
True
```

Type promotion

CuPy automatically promotes dtypes of `cupy.ndarray` in a function with two or more operands, the result dtype is determined by the dtypes of the inputs. This is different from NumPy's rule on type promotion, when operands contain zero-dimensional arrays. Zero-dimensional `numpy.ndarray` are treated as if they were scalar values if they appear in operands of NumPy's function. This may affect the dtype of its output, depending on the values of the "scalar" inputs.

```
>>> (np.array(3, dtype=np.int32) * np.array([1., 2.], dtype=np.float32)).dtype
dtype('float32')

>>> (cp.array(3, dtype=np.int32) * cp.array([1., 2.], dtype=np.float32)).dtype
dtype('float64')
```

Matrix type (`numpy.matrix`)

SciPy returns `numpy.matrix` (a subclass of `numpy.ndarray`) when dense matrices are computed from sparse matrix + ndarray). However, CuPy returns `cupy.ndarray` for such operations.

There is no plan to provide `numpy.matrix` equivalent in CuPy. This is because the use of `numpy.matrix` is no longer recommended since NumPy 1.15.

Data types

Data type of CuPy arrays cannot be non-numeric like strings or objects.

Universal Functions only work with CuPy array or scalar

Unlike NumPy, Universal Functions in CuPy only work with CuPy array or scalar. They do not accept other objects (e.g., lists or `numpy.ndarray`).

```
>>> np.power([np.arange(5)], 2)
array([[ 0,  1,  4,  9, 16]])

>>> cp.power([cp.arange(5)], 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "cupy/_core/_kernel.pyx", line 1285, in cupy._core._kernel.ufunc.__call__
  File "cupy/_core/_kernel.pyx", line 159, in cupy._core._kernel._preprocess_args
  File "cupy/_core/_kernel.pyx", line 145, in cupy._core._kernel._preprocess_arg
TypeError: Unsupported type <class 'list'>
```

Random seed arrays are hashed to scalars

Like NumPy, CuPy's RandomState objects accept seeds either as numbers or as full NumPy arrays.

However, unlike NumPy, array seeds will be hashed down to a single number of 64 bits. In contrast, NumPy often converts the seeds into a larger state space of 128 bits. Therefore, CuPy's implementation may not communicate as much entropy to the underlying random number generator.

NaN (not-a-number) handling

Prior to CuPy v11, CuPy's reduction functions (e.g., `cupy.sum()`) handle NaNs in complex numbers differently from NumPy's counterparts:

```
>>> a = [0.5 + 3.7j, complex(0.7, np.nan), complex(np.nan, -3.9), complex(np.nan, np.nan)]
>>> a
[(0.5+3.7j), (0.7+nanj), (nan-3.9j), (nan+nanj)]
>>>
>>> a_np = np.asarray(a)
>>> print(a_np.max(), a_np.min())
(0.7+nanj) (0.7+nanj)
>>>
>>> a_cp = cp.asarray(a_np)
>>> print(a_cp.max(), a_cp.min())
(nan-3.9j) (nan-3.9j)
```

The reason is that internally the reduction is performed in a strided fashion, thus it does not ensure a proper comparison order and cannot follow NumPy's rule to always propagate the first-encountered NaN. This difference does not apply when CUB library is enabled which is the default for CuPy v11 and later.

Contiguity / Strides

To provide the best performance, the contiguity of a resulting ndarray is not guaranteed to match with that of

```
>>> a = np.array([[1, 2], [3, 4]], order='F')
>>> a
array([[1, 2],
       [3, 4]])
>>> print((a + a).flags.f_contiguous)
True

>>> a = cp.array([[1, 2], [3, 4]], order='F')
>>> a
array([[1, 2],
       [3, 4]])
>>> print((a + a).flags.f_contiguous)
False
```

Interoperability

CuPy implements standard APIs for data exchange and interoperability, which means it can be used in conjunction with any other libraries supporting the standard. For example, NumPy, Numba, PyTorch, TensorFlow, MPI4Py among others can be directly operated on CuPy arrays.

NumPy

CuPy implements `__array_ufunc__` interface (see NEP 13), `__array_function__` interface (see NEP 18), and other Python Array API Standard.

Note that the return type of these operations is still consistent with the initial type.

```
>>> import cupy as cp
>>> import numpy as np
>>> dev_arr = cp.random.randn(1, 2, 3, 4).astype(cp.float32)
>>> result = np.sum(dev_arr)
>>> print(type(result))
<class 'cupy.ndarray'>
```

> **Note**
>
> `__array_ufunc__` feature requires NumPy 1.13 or later
>
> `__array_function__` feature requires NumPy 1.16 or later. As of NumPy 1.17, `__array_function__` is enabled by default
>
> NEP 13 — A mechanism for overriding Ufuncs
>
> NEP 18 — A dispatch mechanism for NumPy's high level array functions

Numba

Skip to content

`__cuda_array_interface__` which is compatible with Numba v0.39.0 or later Interface for details). It means one can pass CuPy arrays to kernels JITed with Numba.

#################### FIXME: crashes when launching

```
import cupy as cp
from numba import cuda

@cuda.jit
def add(x, y, out):
        start = cuda.grid(1)
        stride = cuda.gridsize(1)
        for i in range(start, x.shape[0], stride):
                out[i] = x[i] + y[i]

a = cp.arange(10)
b = a * 2

out = cp.zeros_like(a)
print(out)  # => [0 0 0 0 0 0 0 0 0 0]

add[1, 32](a, b, out)
print(out)  # => [ 0  3  6  9 12 15 18 21 24 27]
```

In addition, `cupy.asarray()` supports zero-copy conversion from Numba CUDA array to CuPy array.

```
>>> import numpy as np
>>> import cupy as cp
>>> import numba
>>> x = np.arange(10)
>>> type(x)
<class 'numpy.ndarray'>
>>> x_numba = numba.cuda.to_device(x)
>>> type(x_numba)
<class 'numba.cuda.cudadrv.devicearray.DeviceNDArray'>
>>> x_cupy = cp.asarray(x_numba)
>>> type(x_cupy)
<class 'cupy.ndarray'>
```

CPU/GPU agnostic code

Once beginning porting code to the GPU, one has to consider how to handle creating data on either the CPU or GPU. CuPy's compatibility with NumPy/SciPy makes it possible to write CPU/GPU agnostic code. For this purpose, CuPy implements the `cupy.get_array_module()` function that returns a reference to cupy if any of its arguments resides on a GPU and numpy

Here is an example of a CPU/GPU agnostic function

```
>>> import numpy as np
>>> import cupy as cp
>>> # define a simple function: f(x)=x+1
>>> def addone(x):
...     xp = cp.get_array_module(x) # Returns cupy if any array is on the GPU, otherwis
...     print("Using:", xp.__name__)
...     return x+1
>>> # create an array and copy it to GPU
>>> a_cpu = np.arange(0, 20, 2)
>>> a_gpu = cp.asarray(a_cpu)
>>> # GPU/CPU agnostic code also works with CuPy
>>> print(addone(a_cpu))
Using: numpy
[ 1  3  5  7  9 11 13 15 17 19]
>>> print(addone(a_gpu))
Using: cupy
[ 1  3  5  7  9 11 13 15 17 19]
```

User-Defined Kernels

Sometimes you need a specific GPU function or routine that is not provided by an existing library or tool. In these situation, you need to write a “custom kernel”, i.e. a user-defined GPU kernel. Custom kernels written with CuPy only require a small snippet of code and CuPy automatically wraps and compiles it. Compiled binaries are then cached and reused in subsequent runs.

CuPy provides three templates of user-defined kernels:

- cupy.ElementwiseKernel: User-defined elementwise kernel
- cupy.ReductionKernel: User-defined reduction kernel
- cupy.RawKernel: User-defined CUDA/HIP kernel

ElementwiseKernel

The element-wise kernel focuses on kernels that operate on an element-wise basis. An element-wise kernel has four components:

- input argument list
- output argument list
- function code
- kernel name

The argument lists consist of comma-separated argument definitions. Each argument definition

specifier and an argument name. Names of NumPy data types can be used as type specifiers.

```
>>> my_kernel = cp.ElementwiseKernel(
...     'float32 x, float32 y',  # input arg list
...     'float32 z',             # output arg list
...     'z = (x - y) * (x - y)', # function
...     'my_kernel')             # kernel name
```

In the first line, the object instantiation is named `my_kernel`. The next line has the variables to be used as input (x and y) and output (z). These variables can be typed with NumPy data types, as shown. The function code then follows. The last line states the kernel name, which is `my_kernel`, in this case.

The above kernel can be called on either scalars or arrays since the ElementwiseKernel class does the indexing with broadcasting automatically:

```
>>> import cupy as cp
>>> # user-defined kernel
>>> my_kernel = cp.ElementwiseKernel(
...     'float32 x, float32 y',
...     'float32 z',
...     'z = (x - y) * (x - y)',
...     'my_kernel')
>>> # allocating arrays x and y
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> y = cp.arange(5, dtype=np.float32)
>>> # launch the kernel
>>> my_kernel(x,y)
array([[ 0.,  0.,  0.,  0.,  0.],
       [25., 25., 25., 25., 25.]], dtype=float32)
>>> my_kernel(x,5)
array([[25., 16.,  9.,  4.,  1.],
       [ 0.,  1.,  4.,  9., 16.]], dtype=float32)
```

Sometimes it would be nice to create a generic kernel that can handle multiple data types. CuPy allows this with the use of a type placeholder. The above `my_kernel` can be made type-generic as follows:

```
my_kernel_generic = cp.ElementwiseKernel(
   'T x, T y',
   'T z',
   'z = (x - y) * (x - y)',
   'my_kernel_generic')
```

If a type specifier is one character, T in this case, it is treated as a **type placeholder**. Same character in the kernel definition indicates the same type. More than one type placeholder can be used in a kernel definition. The actual type of these placeholders is determined by the actual argument type. The ElementwiseKernel class first checks the output arguments and then the input arguments to determine the actual type. If no output arguments are given on the kernel invocation, only the input arguments are used to determine the type.

```
my_kernel_generic2 = cp.ElementwiseKernel(
    'X x, Y y',
    'Z z',
    'z = (x - y) * (x - y)',
    'my_kernel_generic2')
```

> Note
>
> This above kernel, i.e. `my_kernel_generic2`, requires the output argument to be explicitly specified, because the type Z cannot be automatically determined from the input arguments X and Y.

ReductionKernel

The second type of CuPy custom kernel is the reduction kernel, which is focused on kernels of the Map-Reduce type. The ReductionKernel class has four extra parts:

- Identity value: Initial value of the reduction
- Mapping expression: Pre-processes each element to be reduced
- Reduction expression: An operator to reduce the multiple mapped values. Two special variables, a and b, are used for this operand
- Post-mapping expression: Transforms the resulting reduced values. The special variable a is used as input. The output should be written to the output variable

Here is an example to compute L2 norm along specified axies:

```
>>> import cupy as cp
>>> # user-defined kernel
>>> l2norm_kernel = cp.ReductionKernel(
    'T x',  # input arg list
    'T y',  # output arg list
    'x * x',  # mapping
    'a + b',  # reduction
    'y = sqrt(a)',  # post-reduction mapping function
    '0',  # identity value
    'l2norm'  # kernel name
    )
>>> # allocating array
>>> x = cp.arange(10, dtype=np.float32).reshape(2, 5)
>>> x
array([[0., 1., 2., 3., 4.],
       [5., 6., 7., 8., 9.]], dtype=float32)
>>> # kernel launch
>>> l2norm_kernel(x, axis=1)
array([ 5.477226 , 15.9687195], dtype=float32)
```

RawKernel

The last is the RawKernel class, which is used to define kernels from raw CUDA/HIP source code.

RawKernel object allows you to call the kernel with CUDA's cuLaunchKernel interface, and this gives you control of e.g. the grid size, block size, shared memory size, and stream.

```
>>> add_kernel = cp.RawKernel(r'''
... extern "C" __global__
... void my_add(const float* x1, const float* x2, float* y) {
...     int tid = blockDim.x * blockIdx.x + threadIdx.x;
...     y[tid] = x1[tid] + x2[tid];
... }
... ''', 'my_add')
>>> x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
>>> y = cp.zeros((5, 5), dtype=cp.float32)
>>> add_kernel((5,), (5,), (x1, x2, y))
>>> y
array([[ 0.,  2.,  4.,  6.,  8.],
       [10., 12., 14., 16., 18.],
       [20., 22., 24., 26., 28.],
       [30., 32., 34., 36., 38.],
       [40., 42., 44., 46., 48.]], dtype=float32)
```

> Note
>
> The kernel does not have return values. You need to pass both input arrays and output arrays as arguments.
>
> When using printf() in your GPU kernel, you may need to synchronize the stream to see the output.
>
> The kernel is declared in an extern “C” block, indicating that the C linkage is used. This is to ensure the kernel names are not mangled so that they can be retrieved by name.

`cupy.fuse` decorator

Apart from using the above templates for custom kernels, CuPy provides the `cupy.fuse` decorator which “fuse” the custom kernel functions to a single kernel function, therefore creating a dramatic lowering of the launching overhead. Moreover, the syntax looks like a Numba decorator, it is much easier to define an elementwise or reduction kernel than using the ElementwiseKernel or ReductionKernel template.

However, it is still experimental, i.e. there are bugs and incomplete functionalities to be fixed.

Here is the example using `cupy.fuse()` decorator

```
>>> import cupy as cp
>>> # adding decorator to the function squared_diff
>>> @cp.fuse(kernel_name='squared_diff')
... def squared_diff(x, y):
...     return (x - y) * (x - y)
>>> # allocating x and y
>>> x = cp.arange(10,dtype=cp.float32)
>>> y = x[::-1]
>>> # call the function
>>> squared_diff(x, y)
array([81, 49, 25,  9,  1,  1,  9, 25, 49, 81])
```

Low-level features

In addition to custom kernels, accessing low-level CUDA/HIP features are available for those who need more fine-grain control for performance:

- Stream and Event: CUDA stream and per-thread default stream are supported by all APIs
- Memory Pool: Customizable memory allocator with a built-in memory pool
- Profiler: Supports profiling code using CUDA Profiler and NVTX
- Host API Binding: Directly call CUDA libraries, such as NCCL, cuDNN, cuTENSOR, and cuSPARSELt APIs from Python

Summary

In this episode, we have learned about:

- CuPy basics
- Moving data between the CPU and GPU devices
- Different ways to launch GPU kernels

> Keypoints
>
> - GPUs have massive computing power compared to CPU
> - CuPy is a good first step to start
> - CuPy provides an extensive collection of GPU array functions
> - Always have both the CPU and GPU versions of your code available so that you can compare performance, as well as validate the results
> - Fine-tuning for optimal performance of real-world applications can be tedioius

See also

- CuPy Homepage
- GPU programming: When, Why and How?
- CUDA Python from Nvidia
- CuPy Wiki page
- Chainer Blog

Introduction to HPC

> Questions
>
> - What is High-Performance Computing (HPC)?
> - Why do we use HPC systems?
> - How does parallel computing make programs faster?

> Objectives
>
> - Define what High-Performance Computing (HPC) is.
> - Identify the main components of an HPC system.
> - Describe the difference between serial and parallel computing.

Skip to content

nmand on a cluster using the terminal.

High-Performance Computing (HPC) refers to using many computers working together to solve complex problems faster than a single machine could. HPC is widely used in fields such as climate science, molecular simulation, astrophysics, and artificial intelligence.

This lesson introduces what HPC is, why it matters, and how researchers use clusters to perform large-scale computations.

What is HPC?

HPC systems, often called *supercomputers* or *clusters*, are made up of many computers (called **nodes**) connected by a fast network. Each node can have multiple cores which are **CPUs** (and sometimes **GPUs**) that run tasks in parallel.

Typical HPC Components

Component	Description
Login node	Where you connect and submit jobs
Compute nodes	Machines where your program actually runs
Scheduler	Manages job submissions and allocates resources (e.g. SLURM)
Storage	Shared file system accessible to all nodes

Single core performance optimization

Pure-Python loops are slow because each iteration runs in the Python interpreter. NumPy pushes work into optimized native code (C/C++/BLAS), drastically reducing overhead. Below we compare a Python for loop with NumPy vectorized operations and discuss tips for fair, single-core measurements.

Practice making Python faster on a single CPU.

Copy and paste this code

Skip to content

```python
import os
# (Optional safety if you run this inside Python, must be set BEFORE importing numpy)
os.environ.setdefault("OMP_NUM_THREADS", "1")
os.environ.setdefault("OPENBLAS_NUM_THREADS", "1")
os.environ.setdefault("MKL_NUM_THREADS", "1")
os.environ.setdefault("NUMEXPR_NUM_THREADS", "1")

import numpy as np
from time import perf_counter

def timeit(fn, *args, repeats=5, warmup=1, **kwargs):
    # warmup
    for _ in range(warmup):
        fn(*args, **kwargs)
    # timed runs
    tmin = float("inf")
    for _ in range(repeats):
        t0 = perf_counter()
        fn(*args, **kwargs)
        dt = perf_counter() - t0
        tmin = min(tmin, dt)
    return tmin

# Problem size
N = 10_000_000  # 10 million elements

# Test data (contiguous, fixed dtype)
a = np.random.rand(N).astype(np.float64)
b = np.random.rand(N).astype(np.float64)

# --- 1) Pure-Python loop sum ---
def py_sum(x):
    s = 0.0
    for v in x:         # per-element Python overhead
        s += v
    return s

# --- 2) NumPy vectorized sum ---
def np_sum(x):
    return x.sum()      # dispatches to optimized C/BLAS

# --- 3) Elementwise add then sum (Python loop) ---
def py_add_sum(x, y):
    s = 0.0
    for i in range(len(x)):
        s += x[i] + y[i]
    return s

# --- 4) Elementwise add then sum (NumPy, no temporaries) ---
def np_add_sum_no_temp(x, y):
    # np.add.reduce avoids allocating x+y temporary
    return np.add.reduce([x, y])  # equivalent to sum stacks; see alt below

# Alternative that's typically fastest and clearer:
def np_add_sum_fast(x, y):
    return (x + y).sum()  # may allocate a temporary; fast on many BLAS builds
```



```python
print("Timing on single core (best of 5 runs):")
t_py_sum   = timeit(py_sum, a)
```

```
t_np_sum   = timeit(np_sum, a)
t_py_add   = timeit(py_add_sum, a, b)
t_np_add   = timeit(np_add_sum_fast, a, b)

print(f"Python for-loop sum:          {t_py_sum:8.4f} s")
print(f"NumPy vectorized sum:         {t_np_sum:8.4f} s")
print(f"Python loop add+sum:          {t_py_add:8.4f} s")
print(f"NumPy vectorized add+sum:     {t_np_add:8.4f} s")
```

Execute it and following let us verify the effect on the following modifications:

1. Run the timing script with N = 1_000_000, 5_000_000, 20_000_000.
2. Try float32 vs float64.
3. Switch (a + b).sum() to np.add(a, b, out=a); a.sum() and compare.

Practical tips for single-core speed

- Prefer vectorization: Use array ops (+, , *.sum(), .dot(), np.mean, np.linalg.*) rather than per-element Python loops.
- Control temporaries: Expressions like (a + b + c).sum() may create temporaries. When memory is tight, consider in-place ops (a += b) or reductions (np.add(a, b, out=a); np.add.reduce([…])).
- Use the right dtype: float64 is standard for numerics; float32 halves memory traffic and can be faster on some CPUs/GPUs (but mind precision).
- Preallocate: Avoid growing Python lists or repeatedly allocating arrays inside loops.
- Minimize Python in hot paths: Move heavy math into NumPy calls; keep Python for orchestration only.
- Benchmark correctly: Use large N, pin threads to 1 for fair single-core tests, and report the best of multiple runs after a warmup.

–

Parallel Computing

High-Performance Computing relies on **parallel computing**, splitting a problem into smaller parts that can be executed *simultaneously* on multiple processors.

Instead of running one instruction at a time on one CPU core, parallel computing allows you to run many instructions on many cores or even multiple machines at once.

Parallelism can occur at different levels:

Skip to content

e CPU (multiple cores)

le CPUs (distributed nodes)

- **On specialized accelerators** (GPUs or TPUs)

Shared-Memory Parallelism

In **shared-memory** systems, multiple processor cores share the same memory space. Each core can directly read and write to the same variables in memory.

This is the model used in:

- Multicore laptops and workstations
- *Single compute nodes* on a cluster

Programs use **threads** to execute in parallel (e.g., with OpenMP in C/C++/Fortran or **multiprocessing in Python**).

Keypoints

Advantages:

- Easy communication between threads (shared variables)
- Low latency data access

Limitations:

- Limited by the number of cores on one machine
- Risk of race conditions if data access is not synchronized

Practice with threaded parallelism in Python

Example:

```
from multiprocessing import Pool

def square(x):
    return x * x

if __name__ == "__main__":
    with Pool(4) as p:
        result = p.map(square, range(8))
    print(result)
```

Distributed-Memory Parallelism

In distributed-memory systems, each processor (or node) has its own local memory. Processors communicate by passing messages over a network.

Skip to content used when a computation spans multiple nodes in an HPC cluster.

Programs written with MPI (Message Passing Interface) use explicit communication. Below is an example using the Python library `mpi4py` that implements MPI functions in Python

Practice with a simple MPI program

```python
# hello_mpi.py
from mpi4py import MPI

# Initialize the MPI communicator
comm = MPI.COMM_WORLD

# Get the total number of processes
size = comm.Get_size()

# Get the rank (ID) of this process
rank = comm.Get_rank()

print(f"Hello from process {rank} of {size}")

# MPI is automatically finalized when the program exits,
# but you can call MPI.Finalize() explicitly if you prefer
```

For now, do not worry about understanding this code, we will see `mpi4py` in detail later.

Keypoints

Advantages:

- Scales to thousands of nodes
- Each process works independently, avoiding memory contention

Limitations:

- Requires explicit communication (send/receive)
- More complex programming model
- More latency, requires minimizing movement of data.

Hybrid Architectures: CPU, GPU, and TPU

Modern High-Performance Computing (HPC) systems rarely rely on CPUs alone.
They are **hybrid architectures**, combining different types of processors, typically **CPUs**, **GPUs**, and increasingly **TPUs**, to achieve both flexibility and high performance.

CPU: The General-Purpose Processor

Central Processing Units (CPUs) are versatile processors capable of handling a wide range of tasks.

Skip to content

small number of powerful cores optimized for complex, sequential operations

CPUs are responsible for:

- Managing input/output operations
- Coordinating data movement and workflow
- Executing serial portions of applications

They excel in **task parallelism**, where different cores perform distinct tasks concurrently.

GPU: The Parallel Workhorse

Graphics Processing Units (GPUs) contain thousands of lightweight cores that can execute the same instruction on many data elements simultaneously.
This makes them ideal for **data-parallel** workloads, such as numerical simulations, molecular dynamics, and deep learning.

GPUs are optimized for:

- Large-scale mathematical computations
- Highly parallel tasks such as matrix and vector operations

Common GPU computing frameworks include CUDA, HIP, OpenACC, and SYCL.

GPUs provide massive computational throughput but require explicit management of data transfers between CPU and GPU memory.
They are now a standard component of most modern supercomputers.

TPU: Specialized Processor for Tensor Operations

Tensor Processing Units (TPUs) are specialized hardware accelerators designed for tensor and matrix operations, the building blocks of deep learning and AI.
Originally developed by Google, TPUs are now used in both cloud and research HPC environments.

TPUs focus on **tensor computations** and achieve very high performance and energy efficiency for machine learning workloads.
They are less flexible than CPUs or GPUs but excel in neural network training and inference.

Python in High-Performance Computing

Python has become one of the most widely used languages in scientific computing due to its simplicity, readability, and extensive ecosystem of numerical libraries.
Although Python itself is interpreted and slower than compiled languages such as C or Fortran, it now provides a mature set of tools that allow code to **run efficiently on modern HPC architectures**.

These tools map directly to the three fundamental forms of parallelism introduced earlier:

HPC Parallelism Type	Hardware Context	Python Solutions
Shared-memory parallelism	Multicore CPUs within a node	NumPy, Numba, Pythran
Distributed-memory parallelism	Multiple nodes across a cluster	mpi4py
Accelerator parallelism	GPUs and TPUs	CuPy, JAX, Numba (CUDA)

In practice, these technologies allow Python programs to scale from a single core to thousands of nodes on hybrid CPU–GPU systems.

Shared-Memory Parallelism (Multicore CPUs)

Shared-memory parallelism occurs within a single compute node, where all CPU cores access the same physical memory.
Python supports this level of performance primarily through **compiled numerical libraries** and **JIT (Just-In-Time) compilation**, which transform slow Python loops into efficient native machine code.

NumPy: Foundation of Scientific Computing

NumPy provides fast array operations implemented in C and Fortran.
Its vectorized operations and BLAS/LAPACK backends **automatically** exploit shared-memory parallelism through optimized linear algebra kernels.
Although users write Python, most computations occur in compiled native code.

Pythran: Static Compilation of Numerical Python Code

Pythran compiles numerical Python code — particularly code using NumPy — into optimized

It can automatically parallelize loops using **OpenMP**, enabling true multicore utilization without manual thread management.

Key strengths:

- Converts array-oriented Python functions into C++ for near-native speed
- Supports automatic OpenMP parallelization for CPU cores
- Integrates easily into existing Python workflows

Pythran is well-suited for simulations or kernels that need to exploit multiple cores on a node.

Numba: JIT Compilation for Shared and Accelerator Architectures

Numba uses LLVM to JIT-compile Python functions into efficient machine code at runtime.
On multicore CPUs, Numba can parallelize loops using OpenMP-like constructs; on GPUs, it can emit CUDA kernels (see below).

Main advantages:

- Minimal syntax changes required
- Explicit parallel decorators for CPU threading
- Compatible with NumPy arrays and ufuncs

Together, NumPy, Pythran, and Numba enable Python to fully exploit shared-memory parallelism.

Distributed-Memory Parallelism (Clusters and Supercomputers)

At large scale, HPC systems use **distributed memory**, where each node has its own local memory and must communicate explicitly.
Python provides access to this level of parallelism through **mpi4py**, a direct interface to the standard MPI library.

mpi4py: Scalable Distributed Computing with MPI

mpi4py enables Python programs to exchange data between processes running on different nodes using MPI.
It provides both point-to-point and collective communication primitives, identical in concept to those used in C or Fortran MPI applications.

Key features:

- Works seamlessly with NumPy arrays (zero-copy data transfer)
- Supports all MPI operations (send, receive, broadcast, scatter, gather, reduce)

Skip to content h job schedulers such as SLURM or PBS

With `mpi4py`, Python can participate in large-scale distributed-memory simulations or data-parallel tasks across thousands of cores.

Accelerator-Specific Parallelism (GPUs and TPUs)

Modern HPC nodes increasingly include **GPUs** or **TPUs** to accelerate numerical workloads. Python offers several mature libraries that interface directly with these accelerators, providing high-level syntax while executing low-level parallel kernels.

CuPy: GPU-Accelerated NumPy Replacement

CuPy mirrors the NumPy API but executes array operations on GPUs using CUDA (NVIDIA) or ROCm (AMD).
Users can port existing NumPy code to GPUs with minimal changes, gaining massive speedups for large, data-parallel computations.

Highlights:

- NumPy-compatible array and linear algebra operations
- Native support for multi-GPU and CUDA streams
- Tight integration with deep learning and simulation frameworks

JAX: Unified Array Computing for CPUs, GPUs, and TPUs

JAX combines automatic differentiation and XLA-based compilation to execute Python functions efficiently on CPUs, GPUs, and TPUs.
It is particularly well-suited for scientific machine learning and differentiable simulations.

Key strengths:

- Just-In-Time (JIT) compilation via XLA
- Transparent execution on accelerators (GPU, TPU)
- Built-in vectorization and automatic differentiation

JAX provides a single high-level API for heterogeneous HPC nodes, seamlessly handling hybrid CPU–GPU–TPU workflows.

Summary: Python Across HPC Architectures

Python can now leverage **all layers of hybrid HPC architectures** through specialized libraries:

Architecture	Parallelism Type	Typical Python Tools	Example Use Cases
Multicore CPUs	Shared memory	NumPy, Pythran, Numba	Numerical kernels, vectorized math
Clusters	Distributed memory	mpi4py	Large-scale simulations, domain decomposition
GPUs / TPUs	Accelerator parallelism	CuPy, JAX, Numba (CUDA)	Machine learning, dense linear algebra

Together, these tools allow Python to serve as a *high-level orchestration language* that transparently scales from a single laptop core to full supercomputing environments — integrating shared-memory, distributed-memory, and accelerator-based parallelism in one ecosystem.

Keypoints

- Python's ecosystem maps naturally onto hybrid HPC architectures.
- **NumPy, Numba, and Pythran** exploit shared-memory parallelism on multicore CPUs.
- **mpi4py** extends Python to distributed-memory clusters.
- **CuPy and JAX** enable acceleration on GPUs and TPUs.
- These libraries allow researchers to combine high productivity with near-native performance across all layers of HPC systems.

Introduction to MPI with Python (mpi4py)

Questions

- What is MPI, and how does it enable parallel programs to communicate?
- How does Python implement MPI through the `mpi4py` library?
- What are point-to-point and collective communications?
- How does `mpi4py` integrate with NumPy for efficient data exchange?

Objectives

- Understand the conceptual model of MPI: processes, ranks, and communication.
- veen point-to-point and collective operations.

- lumPy arrays act as communication buffers in `mpi4py`.
- See now `mpi4py` bridges Python and traditional HPC concepts.

What Is MPI?

MPI (Message Passing Interface) is a standardized programming model for communication among processes that run on **distributed-memory systems**, such as HPC clusters.

In a distributed-memory system, each compute node (or process) has its **own local memory**. Unlike shared-memory systems, where threads can directly read and write to a common address space, distributed processes **cannot directly access each other's memory**.
To collaborate, they must explicitly **send and receive messages** containing the data they need to share.

Independent Processes and the SPMD Model

When you run an MPI program, the system launches **multiple independent processes**, each running its **own copy** of the same program.
This design is fundamental: because each process owns its own memory space, it must contain its own copy of the code to execute its portion of the computation.

Each process:

- Runs the same code but operates on a different subset of the data.
- Is identified by a unique number called its **rank**.
- Belongs to a **communicator**, a group of processes that can exchange messages (most commonly `MPI.COMM_WORLD`).

This model is known as **SPMD: Single Program, Multiple Data**:
a single source program runs simultaneously on many processes, each working on different data.

Why Copies of the Program Are Needed?

Because processes in distributed memory do not share variables or memory addresses, each process must have:

- Its **own copy of the executable code**, and
- Its **own private workspace (variables, arrays, etc.)**.

This independence is crucial for scalability:

- Each process can execute independently without memory contention.
- an scale to thousands of nodes, since no shared memory bottleneck exists.
- it becomes explicit and controllable, ensuring predictable performance on large clusters.

Sharing Data Between Processes

Although memory is not shared, processes can **cooperate** by exchanging information through **message passing**.
MPI defines two main communication mechanisms:

1. **Point-to-point communication**: Data moves **directly** between two processes.
2. **Collective communication**: Data is exchanged among **all processes** in a communicator in a coordinated way.

> Keypoints
>
> - **Process:** Each MPI program runs as multiple independent processes, not threads.
> - **Rank:** Every process has a unique identifier (its *rank*) within a communicator, used to distinguish and coordinate them.
> - **Communication:** Processes exchange data explicitly through message passing, either **point-to-point** (between pairs) or **collective** (among groups).
>
> Together, these three ideas form the foundation of MPI's model for parallel computing.

---

mpi4py

`mpi4py` is the standard Python interface to the **Message Passing Interface (MPI)**, the same API used by C, C++, and Fortran codes for distributed-memory parallelism.
It allows Python programs to run on many processes, each with its own memory space, communicating through explicit messages.

Communicators and Initialization

In MPI, all communication occurs through a **communicator**, an object that defines which processes can talk to each other.
When a program starts, each process automatically becomes part of a predefined communicator called `MPI.COMM_WORLD`.

This object represents *all processes* that were launched together by `mpirun` or `srun`.

A typical initialization pattern looks like this:

```python
from mpi4py import MPI

comm = MPI.COMM_WORLD      # Initialize communicator
size = comm.Get_size()     # Total number of processes
rank = comm.Get_rank()     # Rank (ID) of this process

print(f"I am rank {rank} of {size}")
```

Every process executes the same code, but rank and size allow them to behave differently.

Hello world MPI

Copy and paste this code and execute it using `mpirun -n N mpi_hello.py` where `N` is the number of tasks.
Note: Do not put more tasks than the number of cores that your computer has.

```python
# mpi_hello.py
from mpi4py import MPI

# Initialize communicator
comm = MPI.COMM_WORLD

# Get the number of processes
size = comm.Get_size()

# Get the rank (ID) of this process
rank = comm.Get_rank()

# Print a message from each process
print(f"Hello world")
```

This code snippet illustrates how independent processes run copies of the program.
To practice further try the following:

1. Use the `rank` variable to print the square of `rank` in each rank.
2. Make the program print only in rank 0, hint: `if rank == 0:`

Solution

Solution 1: Print the square of each rank

```
# mpi_hello_square.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

# Each process prints its rank and the square of its rank
print(f"Process {rank} of {size} has value {rank**2}")
```

Solution 2: Print only one process (rank 0)

```
# mpi_hello_rank0.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

if rank == 0:
    print(f"Hello world from the root process (rank {rank}) out of {size} total processes")
```

Method Naming Convention

In `mpi4py`, most MPI operations exist in **two versions**, a *lowercase* and an *uppercase* form, that differ in how they handle data.

Convention	Example	Description
Lowercase methods	`send()`, `recv()`, `bcast()`, `gather()`	High-level, Pythonic methods that can send and receive arbitrary Python objects. Data is automatically serialized (pickled). Simpler to use but slower for large data.
Uppercase methods	`Send()`, `Recv()`, `Bcast()`, `Gather()`	Low-level, performance-oriented methods that operate on **buffer-like objects** such as NumPy arrays. Data is transferred directly from memory without serialization, achieving near-C speed.

Rule of thumb:

Use *lowercase* methods for small control messages or Python objects,

`ethods for numerical data stored in arrays when performance matters.

Skip to content

`nces

Lowercase (Python objects):

```
comm.send(obj, dest=1)
data = comm.recv(source=0)
```

- The message (obj) can be any Python object.
- MPI automatically serializes and deserializes it internally.
- Fewer arguments: simple but less efficient for large data.

Uppercase (buffer-like objects, e.g., NumPy arrays):

```
comm.Send([array, MPI.DOUBLE], dest=1)
comm.Recv([array, MPI.DOUBLE], source=0)
```

- Requires explicit definition of the data buffer and its MPI datatype. (same syntax as C++)
- Works directly with the memory address of the array (no serialization).
- Achieves maximum throughput for numerical and scientific workloads.

Point-to-Point Communication

The most basic form of communication in MPI is **point-to-point**, meaning data is sent from one process directly to another.

Each message involves:

- A **sender** and a **receiver**
- A **tag** identifying the message type
- A **data buffer** that holds the information being transmitted

These operations are methods of the class `MPI.COMM_WORLD`. This means that one needs to initialize it

Typical operations:

- **Send:** one process transmits data.
- **Receive:** another process waits for that data.

In `mpi4py`, each of these operations maps directly to MPI's underlying mechanisms but with a simple Python interface.

Skip to content allows one process to hand off a message to another in a fully parallel

Examples of conceptual use cases:

- Distributing different chunks of data to multiple workers.
- Passing boundary conditions between neighboring domains in a simulation.

Point-to-Point Communication

Copy and paste the code below into a file called `mpi_send_recv.py`.

```python
# mpi_send_recv.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    # Process 0 sends a message
    data = "Hello from process 0"
    comm.send(data, dest=1)  # send to process 1
    print(f"Process {rank} sent data: {data}")

elif rank == 1:
    # Process 1 receives a message
    data = comm.recv(source=0)  # receive from process 0
    print(f"Process {rank} received data: {data}")

else:
    # Other ranks do nothing
    print(f"Process {rank} is idle")
```

Run the program using: `mpirun -n 3 python mpi_send_recv.py` You should see output indicating that process 0 sent data and process 1 received it, while all others remained idle. Now try:

1. Change the roles: Make process 1 send a reply back to process 0 after receiving the message. Use `comm.send()` and `comm.recv()` in both directions.
2. Blocking communication: Notice that `comm.send()` and `comm.recv()` are blocking operations.

- Add a short delay using `time.sleep(rank)` before sending or receiving.
- Observe how process 0 must wait until process 1 calls `recv()` before it can continue, and vice versa.
- Try swapping the order of the calls (e.g., both processes call `send()` first), what happens?
- You may notice the program hangs or deadlocks, because both processes are waiting for a `recv()` that never starts.

Solution

Solution 1: Change the roles (reply back):

```
# mpi_send_recv_reply.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data_out = "Hello from process 0"
    comm.send(data_out, dest=1)
    print(f"Process {rank} sent: {data_out}")

    data_in = comm.recv(source=1)
    print(f"Process {rank} received: {data_in}")

elif rank == 1:
    data_in = comm.recv(source=0)
    print(f"Process {rank} received: {data_in}")

    data_out = "Reply from process 1"
    comm.send(data_out, dest=0)
    print(f"Process {rank} sent: {data_out}")

else:
    print(f"Process {rank} is idle")
```

Solution 2: Blocking communication behavior:

1. Add a delay (e.g., time.sleep(rank)) before send/recv and observe that each blocking call waits for its partner. Example:

```
# mpi_blocking_delay.py
from mpi4py import MPI
import time

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

time.sleep(rank)  # stagger arrival

if rank == 0:
    comm.send("msg", dest=1)
    print("0 -> sent")
    print("0 -> got:", comm.recv(source=1))

elif rank == 1:
    print("1 -> got:", comm.recv(source=0))
    comm.send("ack", dest=0)
    print("1 -> sent")
```

Skip to content

ıstration:

```python
# mpi_deadlock.py
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank in (0, 1):
    # Both ranks call send() first -> potential deadlock
    comm.send(f"from {rank}", dest=1-rank)
    # This recv may never be reached if partner is also stuck in send()
    print("received:", comm.recv(source=1-rank))
```

Collective Communication

While point-to-point operations handle pairs of processes, **collective operations** involve all processes in a communicator.
They provide coordinated data exchange and synchronization patterns that are efficient and scalable.

Common collectives include:

- **Broadcast:** One process sends data to all others.
- **Scatter:** One process distributes distinct pieces of data to each process.
- **Gather:** Each process sends data back to a root process.
- **Reduce:** All processes combine results using an operation (e.g., sum, max).

Collectives are conceptually similar to group conversations, where every participant either contributes, receives, or both.
They are essential for algorithms that require sharing intermediate results or aggregating outputs.

Collectives

Let us run this code to see the collectives `bcast` and `gather` in action:

```
# mpi_collectives.py
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# --- Broadcast example ---
if rank == 0:
    data = "Hello from the root process"
else:
    data = None

# Broadcast data from process 0 to all others
data = comm.bcast(data, root=0)
print(f"Process {rank} received: {data}")

# --- Gather example ---
# Each process creates its own message
local_msg = f"Message from process {rank}"

# Gather all messages at the root process (rank 0)
all_msgs = comm.gather(local_msg, root=0)

if rank == 0:
    print("\nGathered messages at root:")
    for msg in all_msgs:
        print(msg)
```

Now try the following:

1. Change the root process: In the broadcast section, change the root process from `rank 0` to `rank 1`.
2. How would be the same done with point to point communication?

Solution

Solution 1: Change the root process: The root process is the one that handles the behaviour of the collectives. So we just need to change the root of the collective **broadcast**.

```
# --- Broadcast example ---
if rank == 1:
    data = "Hello from process 1 (new root)"
else:
    data = None

# Broadcast data from process 1 to all others
data = comm.bcast(data, root=1)
print(f"Process {rank} received: {data}")
```

Solution 2: Manual broadcasting the previous code: To reproduce a broadcast manually using only send() and recv(), one could write:

```
# Manual broadcast using point-to-point
if rank == 1:
    data = "Hello from process 1 (manual broadcast)"
    # Send to all other processes
    for dest in range(size):
        if dest != rank:
            comm.send(data, dest=dest)
else:
    data = comm.recv(source=1)

print(f"Process {rank} received: {data}")
```

Integration with NumPy: Buffer-Like Objects

A major strength of `mpi4py` is its **direct integration with NumPy arrays**.
MPI operations can send and receive **buffer-like objects**, such as NumPy arrays, without copying data between Python and C memory.

Important

Remember that **buffer-like objects** can be used with the **uppercase methods** for avoiding serialization and its time overhead.

Because NumPy arrays expose their internal memory buffer, MPI can access this data directly. This eliminates the need for serialization (no `pickle` step) and allows **near-native C performance** for communication and collective operations.

Conceptually:

- Each NumPy array acts as a **contiguous memory buffer**.
- MPI transfers data directly from this buffer to another process's memory.
- This mechanism is ideal for large numerical datasets, enabling efficient data movement in parallel programs.

This integration makes it possible to:

- Distribute large datasets across processes using **collectives** like `Scatter` and `Gather`.
- Combine results efficiently with operations like `Reduce` or `Allreduce`.
- Seamlessly integrate parallelism into scientific Python workflows.

Collective Operations on NumPy Arrays

In this example, you will see how collective MPI operations distribute and combine large arrays across multiple processes using **buffer-based communication**.

Save the following code as `mpi_numpy_collectives.py` and run it with multiple processes:

```
# mpi_numpy_collectives.py
from mpi4py import MPI
import numpy as np

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

# Total number of elements in the big array (must be divisible by size)
N = 10_000_000

# Only rank 0 creates the full array
if rank == 0:
    big_array = np.ones(N, dtype="float64")  # for simplicity, all ones
else:
    big_array = None

# Each process will receive a chunk of this size
local_N = N // size

# Allocate local buffer on each process
local_array = np.empty(local_N, dtype="float64")

# Scatter the big array from root to all processes
comm.Scatter(
    [big_array, MPI.DOUBLE],       # send buffer (only valid on root)
    [local_array, MPI.DOUBLE],     # receive buffer on all processes
    root=0,
)

# Each process computes a local sum
local_sum = np.sum(local_array)

# Reduce all local sums to a global sum on the root process
global_sum = comm.reduce(local_sum, op=MPI.SUM, root=0)

if rank == 0:
    print(f"Global sum = {global_sum}")
    print(f"Expected   = {float(N)}")
```

Run the program using

```
mpirun -n 4 python mpi_numpy_collectives.py
```

Questions:

distribute and collect data in this program?

2. Why is it necessary to preallocate local_array on every process?
3. What would happen if you used lowercase methods (scatter, reduce) instead of Scatter, Reduce?

Solution

Solution 1: The MPI calls that distribute and collect data in this program are comm.Scatter() and comm.reduce(). Scatter divides the large NumPy array on the root process and sends chunks to all ranks, while Reduce collects the locally computed results and combines them (using MPI.SUM) into a single global result on the root process.

Solution 2: It is necessary to preallocate local_array on every process because the uppercase MPI methods (Scatter, Gather, Reduce, etc.) work directly with memory buffers. Each process must provide a fixed, correctly sized buffer so that MPI can write received data directly into it without additional memory allocation or copying.

Solution 3: If lowercase methods (scatter, reduce) were used instead, MPI would serialize and deserialize the Python objects being communicated (using pickle). This would make the program simpler but significantly slower for large numerical arrays, since it adds extra copying and memory overhead. Using the uppercase buffer-based methods avoids this cost and achieves near-native C performance.

Summary

mpi4py provides a simple yet powerful bridge between Python and the Message Passing Interface used in traditional HPC applications.
Conceptually, it introduces the same communication paradigms used in compiled MPI programs but with Python's expressiveness and interoperability.

Concept	Description
Process	Independent copy of the program with its own memory space
Rank	Identifier for each process within a communicator
Point-to-Point	Direct communication between pairs of processes
Collective	Group communication involving all processes
NumPy Buffers	Efficient memory sharing for large numerical data

mpi4py allows Python users to write distributed parallel programs that scale from laptops to supercomputers, making it an invaluable tool for modern scientific computing.

Keypoints

- MPI creates multiple independent processes running the same program.
- Point-to-point communication exchanges data directly between two processes.
- Collective communication coordinates data exchange across many processes.
- mpi4py integrates tightly with NumPy for efficient, zero-copy data transfers.
- These concepts allow Python programs to scale effectively on HPC systems.

Profiler

Objectives

- Learn how to profile Python code using `cProfile`
- Learn how to visualise cProfile results using `SnakeViz`
- Examine the most most expensive function call via `line_profiler`

Deterministic profilers vs. sampling profilers

While `%timeit` can provide good benchmarking information on single lines or single functions, larger codebases have more complex function hierarchies which require more sofisticated tools to traverse properly.

Deterministic profilers record every function call and event in the program, logging the exact sequence and duration of events.

👍 Pros:

```
- Provides detailed information on the program's execution.

- Deterministic: Captures exact call sequences and timings.
```

👎 Cons:

```
- Higher overhead, slowing down the program.

- Can generate larger amount of data.
```

Sampling profilers periodically samples the program's state (where it is and how much memory is used), providing a statistical view of where time is spent.

```
  - Lower overhead, as it doesn't track every event.

  - Scales better with larger programs.
```

👎 **Cons:**

```
  - Less precise, potentially missing infrequent or short calls.

  - Provides an approximation rather than exact timing.
```

> Note
>
> *Deterministic profilers* are also called *tracing profilers*.

> Discussion
>
> *Analogy*: Imagine we want to optimize the Stockholm Länstrafik (SL) metro system. We wish to detect bottlenecks in the system to improve the service and for this we have asked few passengers to help us by tracking their journey.
>
> - **Deterministic**: We follow every train and passenger, recording every stop and delay. When passengers enter and exit the train, we record the exact time and location.
> - **Sampling**: Every 5 minutes the phone notifies the passenger to note down their current location. We then use this information to estimate the most crowded stations and trains.

Using `cProfile` to investigate performance

Python comes with two built-in tools to profile code, which implement the same interface: `cProfile` and `profile`. These tools can help to identify performance bottlenecks in the code.

In this lesson, we will use `cProfile` due to its smaller overhead (`profile`, on the other hand, is more extensible). The standard syntax to call it is:

```
$ python -m cProfile [-o <outputFile>] <python_module>
```

By default, `cProfile` writes the results to `stdout`, but the optional `-o` flag redirects the output to file instead. A report can be generated using the `pstats` command.

Type-Along

Let's profile the `wordcount` script and write the results to a file.

Warning

Use the shell variant. The profiling output from Jupyter, although it seems to work, is hard to decipher.

IPython / Jupyter | Unix Shell

The `%run` magic supports profiling out-of-the-box using the `-p` flag. The script can be run as:

```
In [1]: %run -p -D wordcount.prof source/wordcount.py data/concat.txt processed_data/concat.dat

*** Profile stats marshalled to file 'wordcount.prof'.
```

Skip to content

Type-Along

Let us consider the following code which simulates a random walk in one dimension. Save it as `walk.py` or download it from `FIXME: here <example/walk.py>`.

Skip to content

```python
"""A 1-D random walk.

See also:
- https://lectures.scientific-python.org/intro/numpy/auto_examples/plot_randomwalk.html

"""
import numpy as np


def step():
    import random
    return 1.0 if random.random() > 0.5 else -1.0


def walk(n: int, dx: float = 1.0):
    """The for-loop version.

    Parameters
    ----------
    n: int
        Number of time steps

    dx: float
        Step size. Default step size is unity.

    """
    xs = np.zeros(n)

    for i in range(n - 1):
        x_new = xs[i] + dx * step()
        xs[i + 1] = x_new

    return xs


def walk_vec(n: int, dx: float = 1.0):
    """The vectorized version of :func:`walk` using numpy functions."""
    import random
    steps = np.array(random.sample([1, -1], k=n, counts=[10 * n, 10 * n]))

    # steps = np.random.choice([1, -1], size=n)

    dx_steps = dx * steps

    # set initial condition to zero
    dx_steps[0] = 0
    # use cumulative sum to replicate time evolution of position x
    xs = np.cumsum(dx_steps)

    return xs


if __name__ == "__main__":
    n = 1_000_000
      = walk(n)
      (n)
```

IPython / Jupyter | Unix Shell

The `%run` magic supports profiling out-of-the-box using the `-p` flag. The script can be run as:

```
In [1]: %run -p -D walk.prof walk.py

*** Profile stats marshalled to file 'walk.prof'.
```

Discussion

Profiling introduces a non-negligible overhead on the code being executed. Thus, the absolute values for time being spent in each function should be taken with a grain of salt. The real objective lies in understanding the *relative* amount of time spent in each function call.

Using SnakeViz to visualise performance reports

SnakeViz is a browser-based visualiser of performance reports generated by `cProfile`. It is already included among the dependecies installed in this virtual/Conda environment.

Type-Along

IPython / Jupyter | Unix Shell

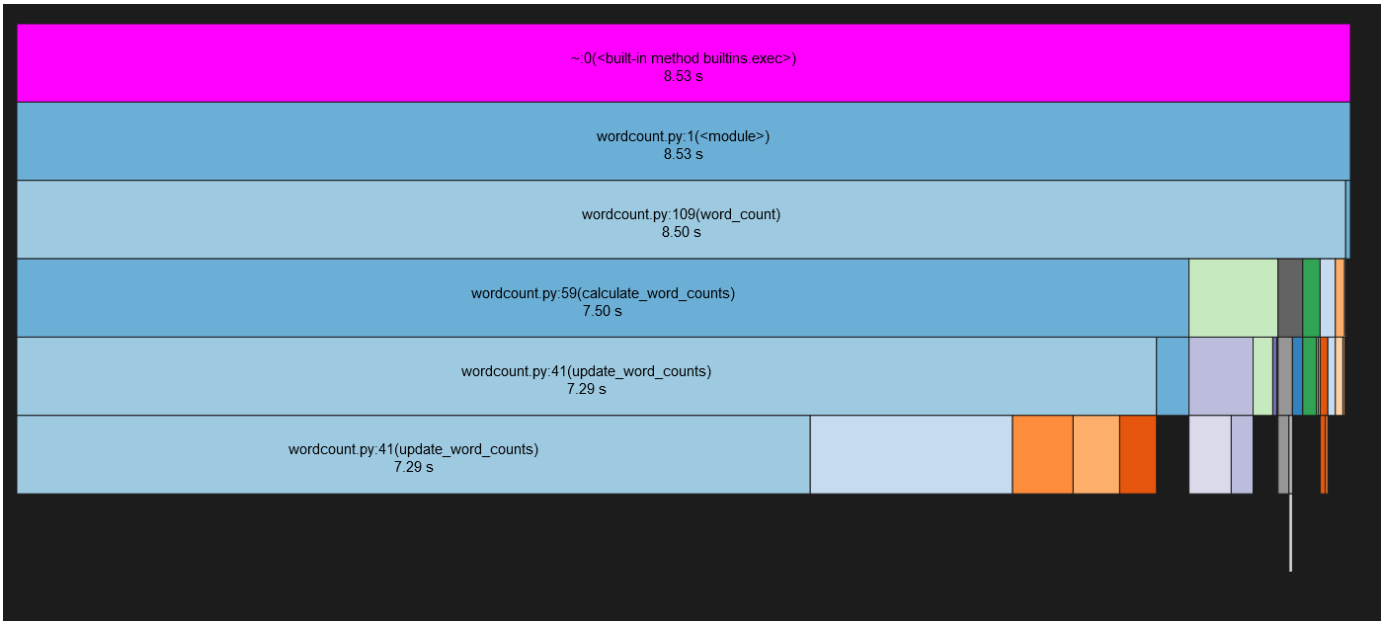
SnakeViz has a IPython magic to profile and open a browser directly. To use it, we just need to load the relevant extension and run it:

```
In [4]: %load_ext snakeviz
In [5]: %snakeviz wordcount.word_count("data/concat.txt", "processed_data/concat.dat", 1)
```

Warning

This will run only if the source IPython instance has access to a local web browser. This also means that, e.g., if you are on Windows and following the tutorial in WSL, this will `not` work.

The output will contain a clickable link containing the visualisation.



Based on the output, we can clearly see that the `update_word_counts()` function is where most of the runtime of the script is spent.

Using `line_profiler` to inspect the expensive function

Once the main performance-intensive function is identified, we can further examine it to find bottlenecks. This can be done using the `line_profiler` tool, which returns a line-by-line breakdown of where time is spent.

Type-Along

Let's profile the `wordcount` script and write the results to a file.

IPython / Jupyter | Unix Shell

The `line_profiler` package provides a magic to be used in IPython. First, the magic needs to be loaded:

```
In [1]: %load_ext line_profiler
```

The script can be run with the `%lprun` magic, whose syntax is very close to the `%run` introduced above. Notice that we have to explicitly mention which functions we want to step through line by line:

```
In [5]: %lprun -f wordcount.update_word_counts wordcount.word_count("data/concat.txt", "processed_c
```

```
Wrote profile results to wordcount.py.lprof
Timer unit: 1e-06 s

Total time: 12.2802 s
File: source/wordcount.py
Function: update_word_counts at line 40

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    40                                           @profile
    41                                           def update_word_counts(line, counts):
    42                                               """
    43                                               Given a string, parse the string and update a dicti
    44                                               counts (mapping words to counts of their frequencie
    45                                               removed before the string is parsed. The function i
    46                                               and words in the dictionary are in lower-case.
    47                                               """
    48  33302070    2574252.9      0.1     21.0      for purge in DELIMITERS:
    49  32068660    4405499.9      0.1     35.9          line = line.replace(purge, " ")
    50   1233410     392268.8      0.3      3.2      words = line.split()
    51   8980773     819407.4      0.1      6.7      for word in words:
    52   7747363    1457841.0      0.2     11.9          word = word.lower().strip()
    53   7747363    1355462.5      0.2     11.0          if word in counts:
    54   7364810    1211000.7      0.2      9.9              counts[word] += 1
    55                                               else:
    56    382553      64505.7      0.2      0.5              counts[word] = 1
```

Based on the output, we can conclude that most of the time is spent replacing delimiters.

Keypoints

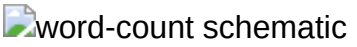
- The `cProfile` module can provide information on how costly each function call is.
- Profile reports can be inspected using the `pstats` tool in tabular form or with SnakeViz for a graphical visualisation
- The `line_profiler` tool can be used to inspect line-by-line performance overhead.

Benchmark

Objectives

- Introduce the example problem
- Preparing the system for benchmarking using `pyperf`
- Learn how to runing benchmarks using `time`, `timeit` and `pyperf timeit`

The problem: word-count-hpda



In this episode, we will use an example project which finds most frequent words in books and plots the result from those statistics. The project contains a script `source/wordcount.py` which is executed to analyze word frequencies from some books. The books are saved in plain-text format in the data directory.

For example to run this code for one book, *pg99.txt*

```
$ git clone https://github.com/ENCCS/word-count-hpda.git
$ cd word-count-hpda
$ python source/wordcount.py data/pg99.txt processed_data/pg99.dat
$ python source/plotcount.py processed_data/pg99.dat results/pg99.png
```

Preparation: Use `pyperf` to tune your system

Most personal laptops would be running in a power-saver / balanced power management mode. This would include that the system has a scaling governor which can change the CPU clock frequency on demand, among other things. This can cause **jitter** which means that benchmarks are not reproducible enough and are less reliable.

In order to improve reliability of your benchmarks consider running the following

Skip to content

It requires admin / root privileges.

```
# python -m pyperf system tune
```

When you are done with the lesson, you can run `python -m pyperf system reset` or restart the computer to go back to your default CPU settings.

See also

- https://pyperf.readthedocs.io/en/latest/system.html#operations-and-checks-of-the-pyperf-system-command
- https://pyperf.readthedocs.io/en/latest/run_benchmark.html#how-to-get-reproducible-benchmark-results
- https://pyperformance.readthedocs.io/usage.html#how-to-get-stable-benchmarks

Benchmark using `time`

In order to observe the cost of computation, we need to choose a sufficiently large input data file and time the computation. We can do that by concatenating all the books into a single input file approximately 45 MB in size.

Type-Along

IPython / Jupyter

Copy the following script.

```
import fileinput
from pathlib import Path

files = Path("data").glob("pg*.txt")
file_concat = Path("data", "concat.txt")

with (
    fileinput.input(files) as file_in,
    file_concat.open("w") as file_out
):
    for line in file_in:
        file_out.write(line)
```

Open an IPython console or Jupyterlab, with `word-count-hpda` as the current working directory (you can also use `%cd` inside IPython to change the directory).

```
%paste

%ls -lh data/concat.txt

import sys
sys.path.insert(0, "source")

import wordcount

%time wordcount.word_count("data/concat.txt", "processed_data/concat.dat", 1)
```

Unix Shell

```
$ cat data/pg*.txt > data/concat.txt
$ ls -lh data/concat.txt
$ time python source/wordcount.py data/concat.txt processed_data/concat.dat
```

Solution

IPython / Jupyter

```
In [1]: %paste
import fileinput
from pathlib import Path

files = Path("data").glob("pg*.txt")
file_concat = Path("data", "concat.txt")

with (
    fileinput.input(files) as file_in,
    file_concat.open("w") as file_out
):
    for line in file_in:
        file_out.write(line)
## -- End pasted text --

In [2]: %ls -lh data/concat.txt
-rw-rw-r-- 1 ashwinmo ashwinmo 45M sep 24 14:54 data/concat.txt

In [3]: import sys
   ...: sys.path.insert(0, "source")

In [4]: import wordcount

In [5]: %time wordcount.word_count("data/concat.txt", "processed_data/concat.dat", 1)
CPU times: user 2.64 s, sys: 146 ms, total: 2.79 s
Wall time: 2.8 s
```

Unix Shell

```
$ cat data/pg*.txt > data/concat.txt
$ ls -lh data/concat.txt
-rw-rw-r-- 1 ashwinmo ashwinmo 46M sep 24 14:58 data/concat.txt
$ time python source/wordcount.py data/concat.txt processed_data/concat.dat

real    0m2,826s
user    0m2,645s
sys     0m0,180s
```

Note

What are the implications of this small benchmark test?

It takes a few seconds to analyze a 45 MB file. Imagine that you are working in a library and you are tasked with running this on several terabytes of data.

- 10 TB = 10 000 000 MB

ng speed = 45 MB / 2.8 s ~ 16 MB/s

: 10 000 000 / 16 = 625 000 s = 7.2 days

Then the same script would take days to complete!

Benchmark using `timeit`

If you run the `%time` magic / `time` command again, you will notice that the results vary a bit. To get a **reliable** answer we should repeat the benchmark several times using `timeit`. `timeit` is part of the Python standard library and it can be imported in a Python script or used via a command-line interface.

If you're using IPython / Jupyter notebook, the best choice will be to use the `%timeit` magic.

As an example, here we benchmark the Numpy array:

```
import numpy as np

a = np.arange(1000)

%timeit a ** 2
# 1.4 µs ± 25.1 ns per loop
```

We could do the same for the `word_count` function.

IPython / Jupyter

```
In [6]: %timeit wordcount.word_count("data/concat.txt", "processed_data/concat.dat", 1)
# 2.81 s ± 12.2 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Unix Shell

We could use `python -m timeit` which is the CLI interface of the standard library module `timeit`,

```
$ export PYTHONPATH=source
$ python -m timeit --setup 'import wordcount' 'wordcount.word_count("data/concat.txt",
1 loop, best of 5: 2.75 sec per loop
```

or an even better alternative is using `python -m pyperf timeit`

```
$ export PYTHONPATH=source
$ python -m pyperf timeit --fast --setup 'import wordcount' 'wordcount.word_count("data
.....
Mean +- std dev: 2.72 sec +- 0.22 sec
```

Notice that the output reports the **arithmetic mean and standard deviation** of timings. This is a good choice, since it means that **outliers and temporary spikes in results are not automatically removed**, which could be as a result of:

- garbage collection
- JIT compilation
- CPU or memory resource limitations

> Keypoints
>
> - `pyperf` can be used to tune the system
> - We understood the use of `time` and `timeit` to create benchmarks
> - `time` is faster, since it is executed only once
> - `timeit` is more reliable, since it collects statistics

Quick Reference

Instructor's guide

Why we teach this lesson

Intended learning outcomes

Timing

Preparing exercises

e.g. what to do the day before to set up common repositories.

Other practical aspects

Interesting questions you might get

Typical pitfalls

Learning outcomes

FIXME

By the end of this module, learners should:

- …
- …

See also

> ⚠ Credit
>
> FIXME
>
> Don’t forget to check out additional course materials from …

License

CC BY-SA for media and pedagogical material

Skip to content

MIT for source code and code snippets

MIT License