Evaluate the performance of this pure python file and then use Numba's jit decorator with the nopython=True and evaluate the speedup of the performance. Be aware of the need to add a dummy function call to let the just-in-time compiler generate a compiled version. Add parallel=True, as well as Numba's prange function to implement the numerical integration in a parallelized manner. Evaluate the speedup and efficiency of the parallelized implementation for a number of processors including 1, 2, 4, 8, 16 and 20 using the NUMBA_NUM_THREADS environment variable. Generate a plot with the speedup and efficiency results, and write the results to the report. Code : "import numpy as np import time def myfunct(x): """ Defines the function to be integrated. INPUTS: - x: double, evaluation point. OUTPUTS: - double, evaluated function. """ return np.sin(x*x)+x/2 # end function def integral_riemann(a,b,N): """ Implements the Riemann integration for the function myfunct(x). INPUTS: - a: double, Lower integration limit. - b: double, Upper integration limit. - N: Int, Number of integration regions. OUTPUTS: - double, evaluated integral. """ dx = (b-a)/N F = 0 for i in range(N): x = a + i*dx F += myfunct(x)*dx # end for return F # end function if __name__ == '__main__': # If needed, add dummy call to the integral_riemann # function here # Evaluate the CPU time and integration here. t_start = time.time() a = 0 b = 2 N = 100_000_000 # 10**8 F = integral_riemann(a,b,N) t_end = time.time() print('Integral {0:f}'.format(F)) print('CPU time:{0:.6f}s'.format(t_end-t_start)) "

Copilot

Certainly! Let's evaluate the performance of the given Python code and then enhance it using Numba for better execution speed. We'll follow these steps:

1. Evaluate the Pure Python Code:
    - First, we'll run the provided Python code without any optimizations.
    - We'll measure the CPU time taken for the integration.
2. Apply Numba Optimization:

        - Next, we’ll use Numba’s JIT (Just-In-Time) compilation to improve performance.
        - We’ll add a dummy function call to ensure Numba generates a compiled version.
        - We’ll also parallelize the numerical integration using Numba’s prange and set nopython=True.
3. Evaluate Speedup and Efficiency:
    - We’ll compare the performance of the original code and the Numba-optimized version.
    - We’ll measure the execution time for both.
    - Additionally, we’ll vary the number of processors (1, 2, 4, 8, 16, and 20) using the NUMBA_NUM_THREADS environment variable.
    - Finally, we’ll generate a plot with speedup and efficiency results.

Here’s the modified code with Numba optimizations:

Python

```
import numpy as np
import time
from numba import jit, prange


def myfunct(x):
    """
    Defines the function to be integrated.

    INPUTS:
    - x: double, evaluation point.

    OUTPUTS:
    - double, evaluated function.
    """
    return np.sin(x*x) + x/2

@jit(nopython=True, parallel=True)
def integral_riemann(a, b, N):
    """
    Implements the Riemann integration for the function myfunct(x).

    INPUTS:
    - a: double, Lower integration limit.
    - b: double, Upper integration limit.
    - N: Int, Number of integration regions.

    OUTPUTS:
    - double, evaluated integral.
    """
```

```
    dx = (b - a) / N
    F = 0

    for i in prange(N):
        x = a + i * dx
        F += myfunct(x) * dx

    return F

if __name__ == '__main__':
    # Add a dummy call to the integral_riemann function
    dummy_result = integral_riemann(0, 1, 1)

    # Evaluate the CPU time and integration
    t_start = time.time()
    a = 0
    b = 2
    N = 100_000_000  # 10**8
    F = integral_riemann(a, b, N)
    t_end = time.time()

    print('Integral: {0:f}'.format(F))
    print('CPU time: {0:.6f}s'.format(t_end - t_start))
```

AI-generated code. Review and use carefully. More info on FAQ.

Now, let's run this optimized code and compare the results. Feel free to adjust the number of processors using the NUMBA_NUM_THREADS environment variable to observe the impact on performance. 🚀

Learn more

1medium.com2numba.pydata.org3medium.com4numba.pydata.org5numba.pydata.org6github.com 7link.springer.com8numba.readthedocs.io9en.wikipedia.org+5 more

You

Bonus: Cython – Matrix-matrix multiplication. Use Cython to implement a matrix-matrix multiplication (MATMUL) function that takes two numpy arrays (two dimensions, float64 arrays) and performs the matrix multiplication as an iterative algorithm: $\mathbf{C} = \mathbf{AB}$, $\mathbf{A} \in \mathbb{R}n\times m$,$\mathbf{B} \in \mathbb{R}m\times p$, $\mathbf{C} \in \mathbb{R}n\times p$ cij = $\sum aikbkj\ m\ k$=1 Compare it to Numpy's matrix-matrix multiplication, the dot function (i.e. C=np.dot(A,B)). Evaluate the performance of the implementation for multiple sizes of matrices, including: 3x3, 10x10, 100x100 and 1000x1000. Note: Since small matrix multiplication may take a very small time to execute, it is recommended that you evaluate the performance on a loop counting for, say 100 matrix multiplications, and average the total time to execute that loop by the number of iterations. Write the results to the report.

Copilot

Certainly! Let’s start by implementing the matrix-matrix multiplication using Cython and then compare it with NumPy’s built-in matrix multiplication (dot function). We’ll evaluate the performance for various matrix sizes.

First, let’s create a Cython implementation for matrix-matrix multiplication. We’ll follow these steps:

4. Implement the matrix multiplication algorithm iteratively.
5. Compare the performance with NumPy’s np.dot function.
6. Evaluate the execution time for different matrix sizes.

Below is the Cython code for matrix-matrix multiplication:

Python

```python
# Save this code in a file named "matrix_multiplication.pyx"

import numpy as np
cimport numpy as np

# Define the matrix multiplication function
cpdef np.ndarray[np.float64_t, ndim=2] matrix_multiply(np.ndarray[np.float64_t, ndim=2] A,
                                                        np.ndarray[np.float64_t, ndim=2] B):
    cdef int n = A.shape[0]
    cdef int m = A.shape[1]
    cdef int p = B.shape[1]
    cdef np.ndarray[np.float64_t, ndim=2] C = np.zeros((n, p), dtype=np.float64)
    cdef int i, j, k

    for i in range(n):
        for j in range(p):
            for k in range(m):
                C[i, j] += A[i, k] * B[k, j]

    return C
```

AI-generated code. Review and use carefully. More info on FAQ.

Next, we’ll compile the Cython code using the following setup:

Python

```python
# Save this code in a file named "setup.py"

from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("matrix_multiplication.pyx"),
```

```
)
```

AI-generated code. Review and use carefully. More info on FAQ.

Compile the Cython code using the command:
python setup.py build_ext --inplace

Now, let's compare the performance:

Python

```python
import numpy as np
from matrix_multiplication import matrix_multiply
import time

# Create random matrices
n = 1000
m = 500
p = 800
A = np.random.rand(n, m)
B = np.random.rand(m, p)

# Evaluate the performance
num_iterations = 100
start_time = time.time()
for _ in range(num_iterations):
    C_cython = matrix_multiply(A, B)
end_time = time.time()
cython_time = (end_time - start_time) / num_iterations

start_time = time.time()
for _ in range(num_iterations):
    C_numpy = np.dot(A, B)
end_time = time.time()
numpy_time = (end_time - start_time) / num_iterations

print(f"Cython execution time: {cython_time:.6f} seconds")
print(f"NumPy execution time: {numpy_time:.6f} seconds")
```

AI-generated code. Review and use carefully. More info on FAQ.

You can adjust the matrix sizes (n, m, and p) and the number of iterations as needed. The results will show the execution time for both Cython and NumPy matrix multiplication. Remember to average the total time over multiple iterations for accurate performance evaluation.

Feel free to run this code and observe the performance differences! 🚀

Learn more
1medium.com2numba.pydata.org3medium.com4numba.readthedocs.io5numba.pydata.org6numba.pydata.org7github.com8link.springer.com9stackoverflow.com10medium.com11stackoverflow.com12stackoverflow.com13infoworld.com+10 more

You

In file included from matrix_multiplication.c:8545:0:
/home/gug303/.conda/envs/hpc/include/python3.12/internal/pycore_frame.h: In function '_PyFrame_Initialize':
/home/gug303/.conda/envs/hpc/include/python3.12/internal/pycore_frame.h:132:5: error: 'for' loop initial declarations are only allowed in C99 mode for (int i = null_locals_from; i < code->co_nlocalsplus; i++) { ^
/home/gug303/.conda/envs/hpc/include/python3.12/internal/pycore_frame.h:132:5: note: use option -std=c99 or -std=gnu99 to compile your code error: command '/usr/bin/gcc' failed with exit code 1

Copilot

I apologize for the inconvenience. It seems that there is an issue with the compilation of the Cython code due to the use of a for loop with initial declarations, which is not allowed in the default C mode.

To address this, you can modify your Cython code to comply with C99 mode by explicitly specifying the -std=c99 flag during compilation. Here's how you can do it:

7. Create a setup.py file (if you haven't already) to compile your Cython code. Here's an example of a setup.py file:

Python

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("matrix_multiplication.pyx", compiler_directives={'language_level': "3"}),
)
```

AI-generated code. Review and use carefully. More info on FAQ.

8. In your Cython code (matrix_multiplication.pyx), add the following line at the beginning to enable C99 mode:

Python

```
#cython: language_level=3
```

AI-generated code. Review and use carefully. More info on FAQ.

9. Compile your Cython code using the following command:

```
python setup.py build_ext --inplace
```

This should resolve the issue related to the for loop initial declarations. Please try these steps, and let me know if you encounter any further issues or need additional assistance! 😊