FastAPI Learning Retro Report

Sync & Async for the FastAPI Path Function Declaration

Jenny 2024/03/08

Practice repo: https://github.com/JennyShen123/fastapi-benchmarking

Conclusion

FastAPI

- **Asynchronous Support**: When dealing with I/O operations, declare path functions as `async` if certain they will use asynchronous functions. This enables FastAPI to optimize performance. If unsure, declare path functions as synchronous (`sync`), and FastAPI will handle optimization. Incorrect declaration can significantly reduce performance.
- **CPU-bound Tasks**: For CPU-intensive tasks, although it's more complex, combining `ProcessPoolExecutor` with `asyncio.get_running_loop` can greatly enhance performance. FastAPI also provides optimizations even if this combination is not used.

Flask

- **Concurrency**: Flask natively does not support asynchronous tasks or multi-threading. However, using `ProcessPoolExecutor` allows it to handle CPU-bound work with efficiency comparable to FastAPI's asynchronous handling.

Go

- **High Performance**: Go is known for its efficiency. If there are no specific implementation issues, using Go is a solid choice, often offering superior performance compared to Python web frameworks.

Build up the Benchmarking ENV

- wrk - a HTTP benchmarking tool

Basic Usage

```
wrk -t12 -c400 -d30s http://127.0.0.1:8080/index.html
```

This runs a benchmark for 30 seconds, using 12 threads, and keeping 400 HTTP connections open.

Output:

```
Running 30s test @ http://127.0.0.1:8080/index.html
  12 threads and 400 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   635.91us    0.89ms  12.92ms   93.69%
    Req/Sec    56.20k     8.07k   62.00k    86.54%
  22464657 requests in 30.00s, 17.76GB read
Requests/sec: 748868.53
Transfer/sec:    606.33MB
```

- reliable tool (?!

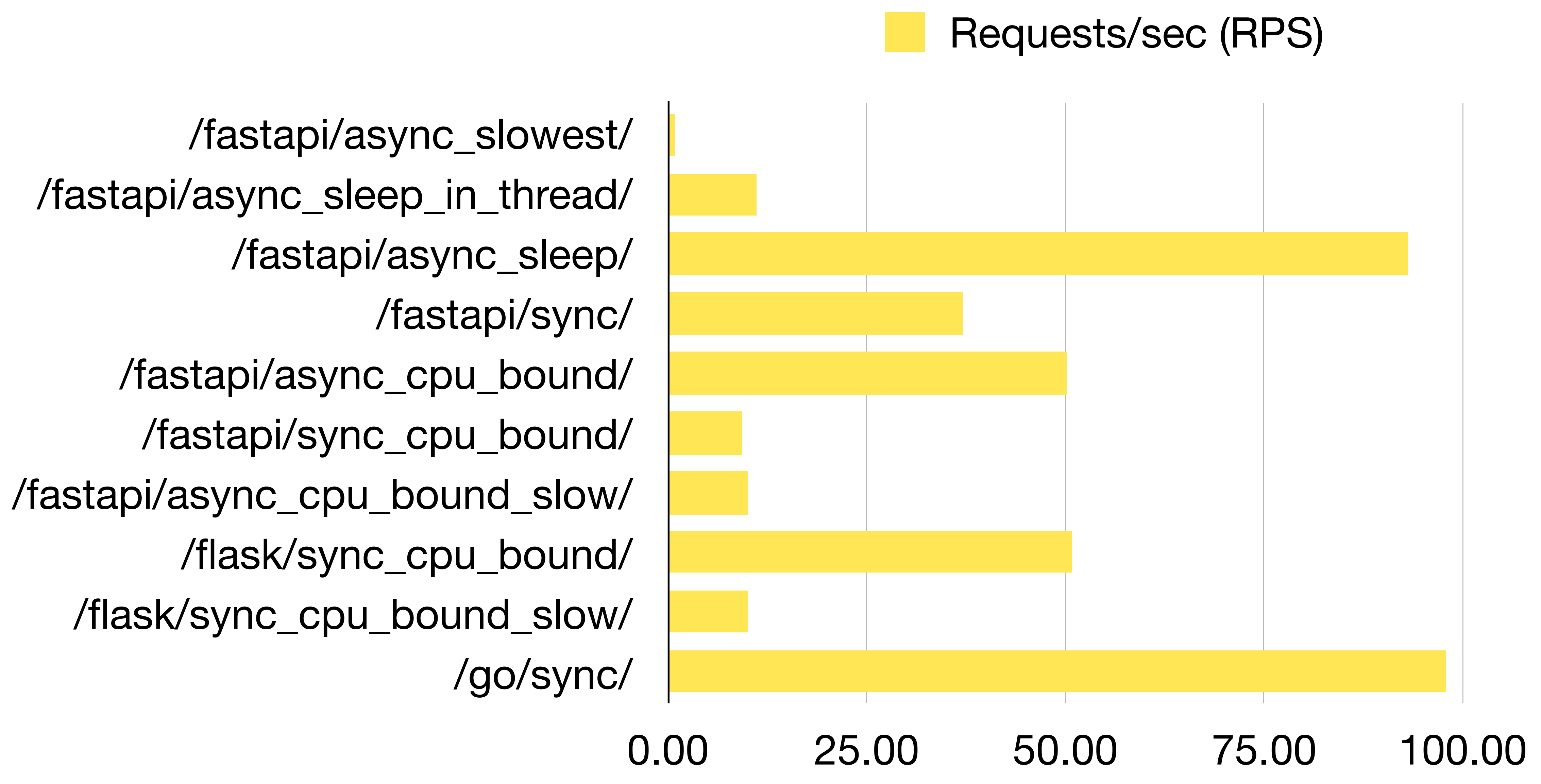
Watch 755 | Fork 2.9k | Star 36.4k

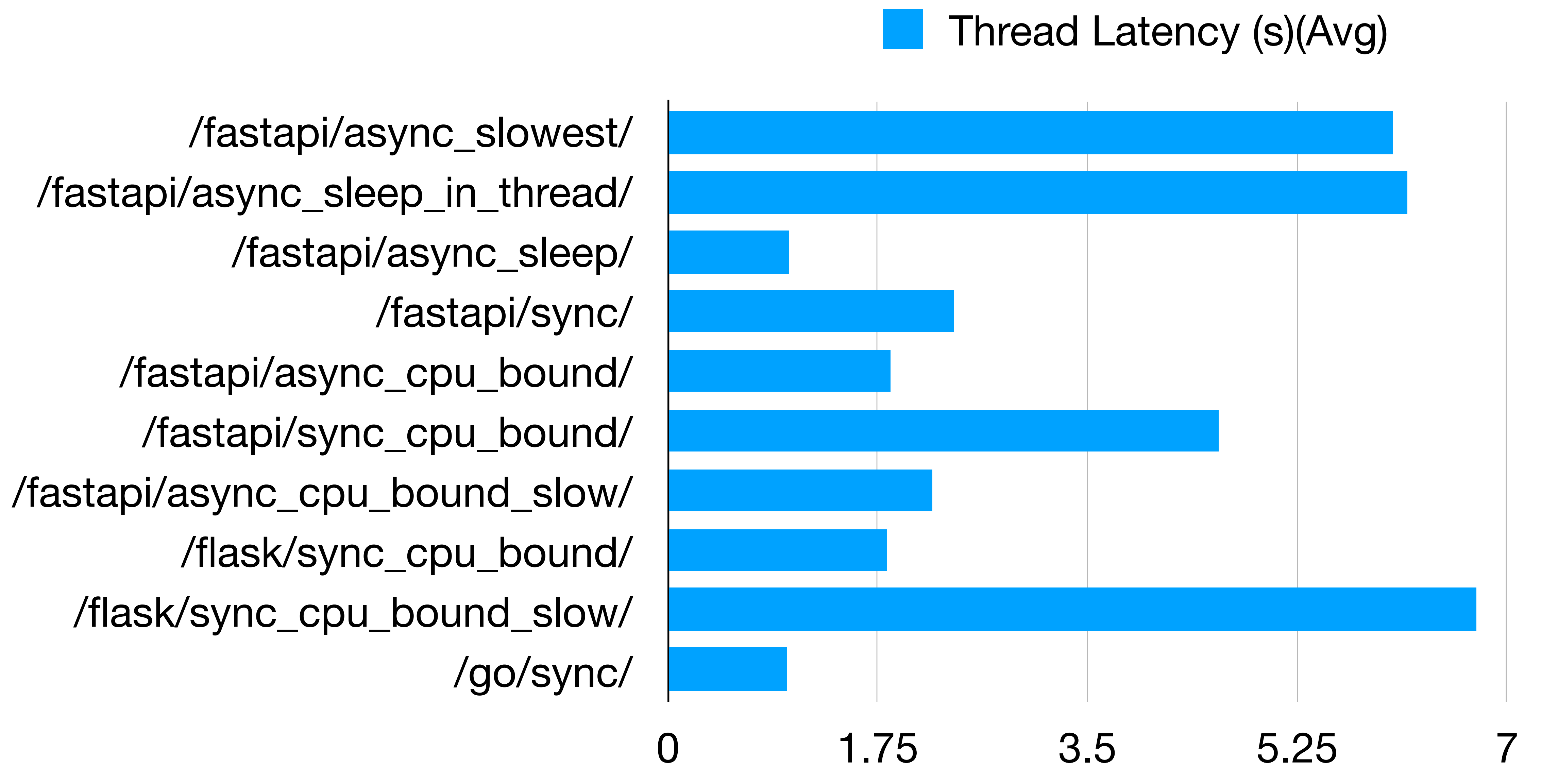
wrk -t4 -c100 -d15s --timeout 100 --latency

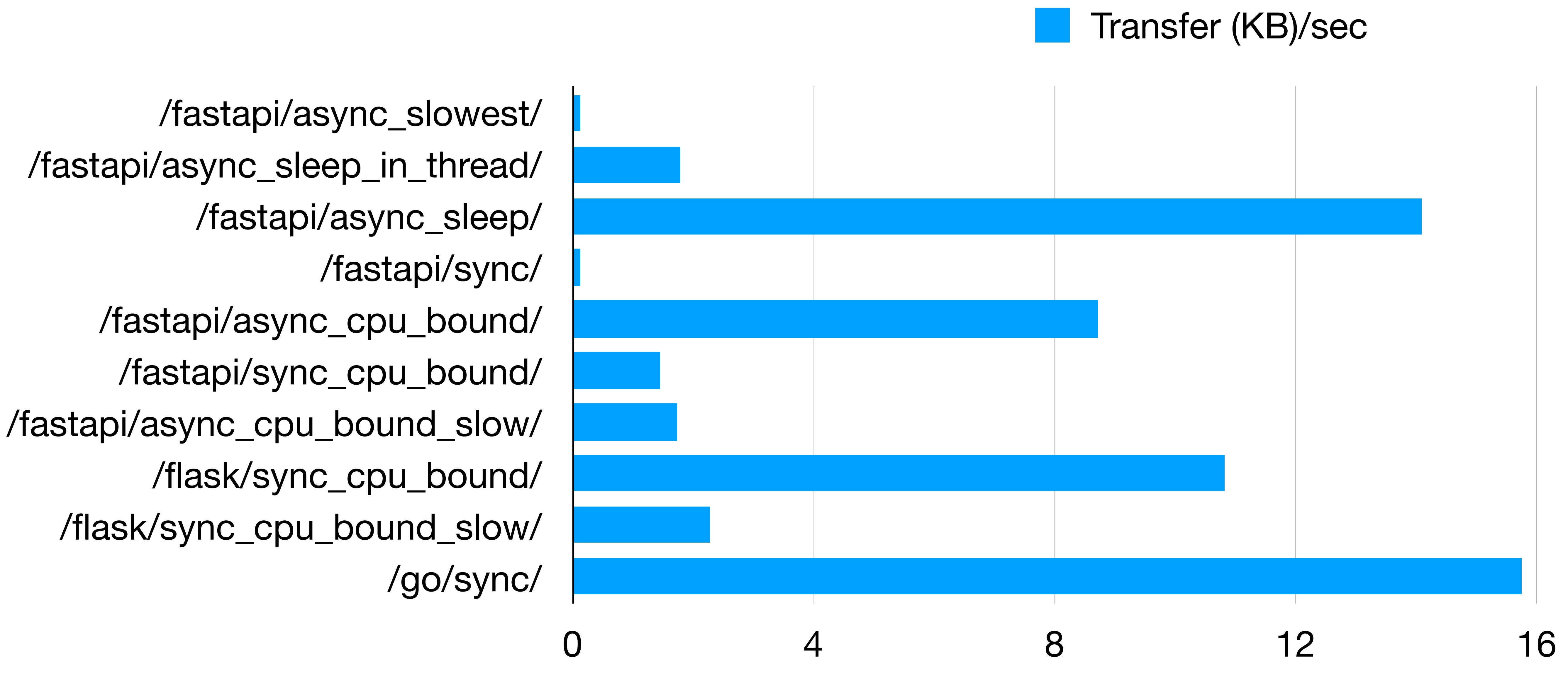
4 threads, 100 connections, for 15 seconds, let the timeout limit 100 seconds

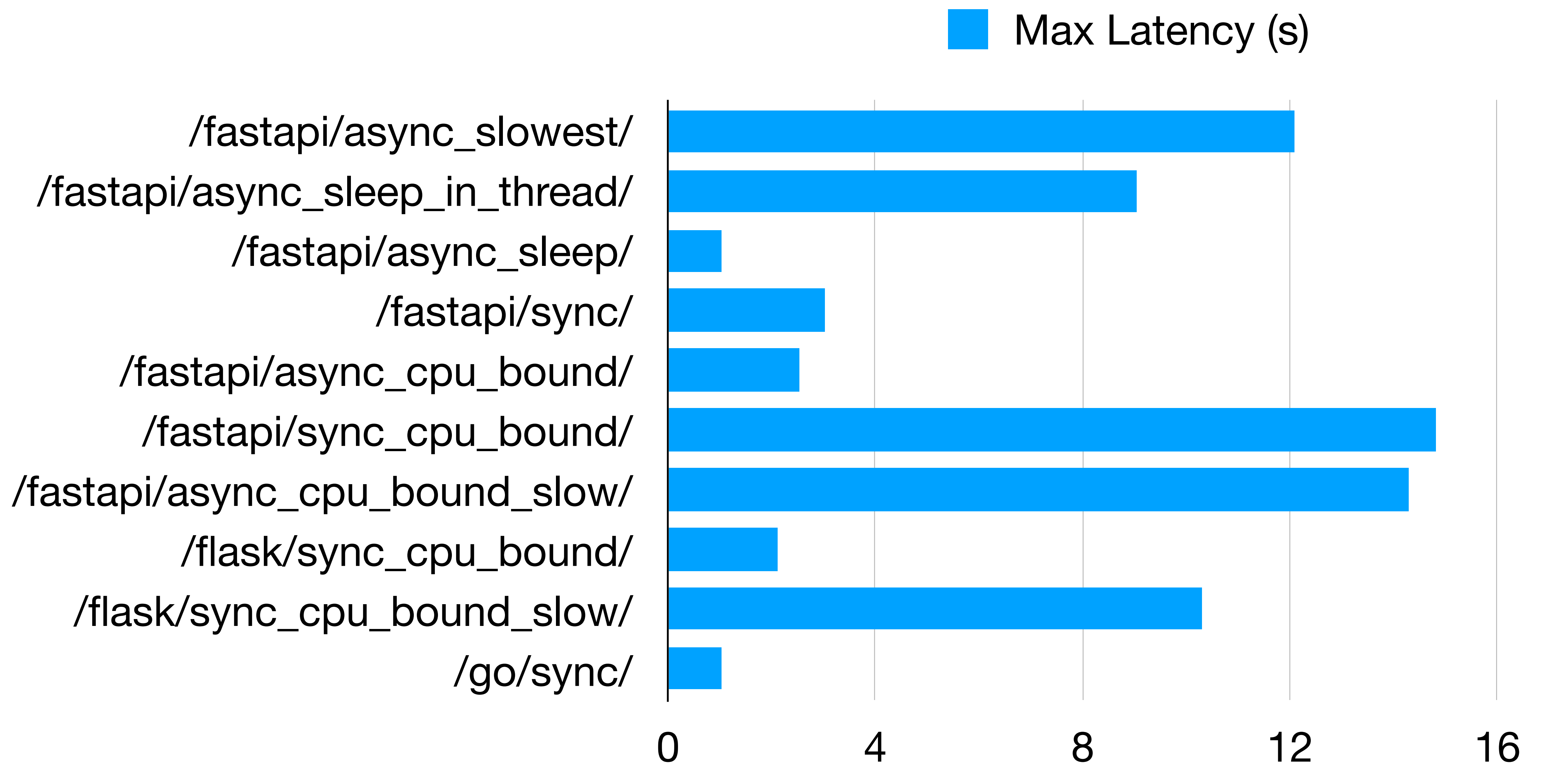
Asyncio particularly suits for I/O-intensive tasks, such as network requests, file I/O, etc. It manages concurrency through event loops and coroutines, allowing you to efficiently perform multiple I/O operations within a single thread.

ProcessPoolExecutor is a high-level interface provided by the concurrent.futures module for parallel execution of multiple processes. It is particularly suitable for CPU-intensive tasks and can utilize multi-core CPUs to accelerate the execution of these tasks.


Requests/sec (RPS)
/fastapi/async_slowest/
/fastapi/async_sleep_in_thread/
/fastapi/async_sleep/
/fastapi/sync/
/fastapi/async_cpu_bound/
/fastapi/sync_cpu_bound/
/fastapi/async_cpu_bound_slow/
/flask/sync_cpu_bound/
/flask/sync_cpu_bound_slow/
/go/sync/
0.00
25.00
50.00
75.00
100.00


Thread Latency (s)(Avg)
/fastapi/async_slowest/
/fastapi/async_sleep_in_thread/
/fastapi/async_sleep/
/fastapi/sync/
/fastapi/async_cpu_bound/
/fastapi/sync_cpu_bound/
/fastapi/async_cpu_bound_slow/
/flask/sync_cpu_bound/
/flask/sync_cpu_bound_slow/
/go/sync/
0
1.75
3.5
5.25
7


Transfer (KB)/sec
/fastapi/async_slowest/
/fastapi/async_sleep_in_thread/
/fastapi/async_sleep/
/fastapi/sync/
/fastapi/async_cpu_bound/
/fastapi/sync_cpu_bound/
/fastapi/async_cpu_bound_slow/
/flask/sync_cpu_bound/
/flask/sync_cpu_bound_slow/
/go/sync/
0
4
8
12
16


Max Latency (s)
/fastapi/async_slowest/
/fastapi/async_sleep_in_thread/
/fastapi/async_sleep/
/fastapi/sync/
/fastapi/async_cpu_bound/
/fastapi/sync_cpu_bound/
/fastapi/async_cpu_bound_slow/
/flask/sync_cpu_bound/
/flask/sync_cpu_bound_slow/
/go/sync/
0
4
8
12
16

Different types of API calls using FastAPI

Code & Explanation	Performance Benchmarking

Code & Explanation

```python
@app.get("/fastapi/async_slowest/")
async def async_slowest():
    time.sleep(1)
    return {"message": "async mode but use sync sleep"}
```

In the given context, `time.sleep(1)` is employed to simulate synchronous I/O operations, which are characterized by not utilizing CPU resources while executing but require a waiting period. This synchronous sleep is placed within a path function declared as asynchronous within the FastAPI framework. The FastAPI framework runs asynchronous functions in an event loop, but without an `await` statement, a function can block, preventing others from running. This results in a sequential execution, similar to synchronous functions.

Performance Benchmarking

```
~/Desktop/practice/fastAPI
 wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:8000/fastapi/async_slowest
Running 15s test @ http://127.0.0.1:8000/fastapi/async_slowest/
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     6.04s     2.68s   12.07s    75.00%
    Req/Sec     0.00      0.00     0.00    100.00%
  Latency Distribution
     50%    6.04s
     75%    7.04s
     90%    8.05s
     99%   12.07s
  12 requests in 15.04s, 1.97KB read
Requests/sec:      0.80
Transfer/sec:     134.00B
```

Code & Explanation

```python
@app.get("/fastapi/async_sleep_in_thread/")
async def async_sleep_in_thread():
    loop = asyncio.get_running_loop()
    await loop.run_in_executor(None, time.sleep, 1)
    return {"message": "sleep run in thread pool"}
```

By employing `asyncio.get_event_loop().run_in_executor`, it can run synchronous functions like `time.sleep(1)` in a separate thread. This approach allows for a mix of synchronous and asynchronous execution, maintaining efficiency. However, each synchronous call occupies a thread, and if there are more requests than available threads, the system can become blocked. This requires managing a balance between context switching and queuing, as excessive requests compete for limited thread resources, possibly leading to delays in task processing until threads become available again.

Performance Benchmarking

```
~/Desktop/practice/fastAPI
 wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:8000/fastapi/async_sleep_in_thread/
Running 15s test @ http://127.0.0.1:8000/fastapi/async_sleep_in_thread/
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     6.17s     2.60s    9.04s    64.29%
    Req/Sec     3.09      5.25    39.00     93.10%
  Latency Distribution
     50%    8.03s
     75%    8.04s
     90%    9.03s
     99%    9.04s
  168 requests in 15.05s, 26.74KB read
Requests/sec:     11.17
Transfer/sec:      1.78KB
```

Different types of API calls using FastAPI

Code & Explanation	Performance Benchmarking

```python
@app.get("/fastapi/async_sleep/")
async def async_sleep():
    await asyncio.sleep(1)
    return {"message": "async mode sleep"}
```

This situation highlights FastAPI's ability to efficiently handle tasks by placing asynchronous functions within async path declarations. FastAPI excels by minimizing context switching. Using `asyncio.sleep(1)` for non-blocking delays showcases the framework's asynchronous capabilities, where task switching is efficiently done within a single thread. This approach avoids the overhead associated with managing multiple OS-level threads, such as thread creation and context switching costs. As a result, FastAPI offers superior execution efficiency by optimizing the use of resources and streamlining task management through asynchronous programming.

```
~/Desktop/practice/fastAPI
 wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:8000/fastapi/async_sleep/
Running 15s test @ http://127.0.0.1:8000/fastapi/async_sleep/
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.01s     4.40ms   1.02s    81.71%
    Req/Sec    56.17     69.69    240.00    81.93%
  Latency Distribution
     50%    1.01s
     75%    1.01s
     90%    1.01s
     99%    1.02s
  1400 requests in 15.04s, 211.91KB read
Requests/sec:     93.10
Transfer/sec:     14.09KB
```

```python
@app.get("/fastapi/sync/")
def sync_sleep():
    time.sleep(1)
    return {"message": "sync, but run in thread pool"}
```

FastAPI automatically runs synchronous functions, like `time.sleep(1)`, in a thread pool via Starlette, avoiding main event loop blockages. This means all synchronous code, regardless of its potential to block, is executed in a thread pool when called from asynchronous routes, ensuring the async event loop runs efficiently. While this automatic process by FastAPI is more efficient than manually managing thread pools in async functions, it remains subject to the constraints of synchronous operations and thread pool capacity.

```
~/Desktop/practice/fastAPI
 wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:8000/fastapi/sync/
Running 15s test @ http://127.0.0.1:8000/fastapi/sync/
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     2.38s   608.83ms   3.03s    50.00%
    Req/Sec    26.53     33.68    120.00    81.48%
  Latency Distribution
     50%    2.03s
     75%    3.02s
     90%    3.02s
     99%    3.03s
  560 requests in 15.04s, 91.33KB read
Requests/sec:     37.24
Transfer/sec:      6.07KB
```

Code & Explanation	Performance Benchmarking

Code & Explanation

```python
def cpu_bound_operation(BIGNUM):
    num = 0
    for i in range(BIGNUM):
        num += 1
    return num
```

```python
@app.get("/fastapi/sync_cpu_bound/")
def sync_cpu_bound():
    BIGNUM = 5000000
    cpu_bound_operation(BIGNUM)
    return {"message": "sync mode operation"}
```

FastAPI also provides basic enhancements for CPU-bound tasks, by utilizing a threading pool. This setup helps in managing resource use more effectively, even for tasks that are traditionally more demanding on system resources.

```python
executor = ProcessPoolExecutor()


async def async_cpu_operation(BIGNUM):
    loop = asyncio.get_running_loop()
    result = await loop.run_in_executor(executor, cpu_bound_operation, BIGNUM)
    return result
```

```python
@app.get("/fastapi/async_cpu_bound/")
async def async_cpu_bound():
    BIGNUM = 5000000
    result = await async_cpu_operation(BIGNUM)
    return {"data": result, "message": "async mode cpu operation"}
```

To further accelerate CPU-bound tasks, which require the CPU to be fully blocked to complete, one can use `asyncio` with `ProcessPoolExecutor` to launch additional workers for computation. This approach is particularly beneficial for CPU-bound tasks, as opposed to I/O tasks, where using `ProcessPoolExecutor` might actually slow down the process due to the overhead of managing multiple processes.

```python
@app.get("/fastapi/async_cpu_bound_slow/")
async def async_cpu_bound_slow():
    BIGNUM = 5000000
    cpu_bound_operation(BIGNUM)
    return {"message": "sync mode operation but run in async mode"}
```

In scenarios involving I/O operations, placing synchronous functions within an asynchronous path function without an `await` can lead to the CPU being blocked, resulting in slower execution. However, for CPU-bound tasks, where blocking the CPU is essential for performing necessary computations, the outcome is comparably similar to synchronous execution.

Performance Benchmarking

```
~/Desktop/practice/fastAPI
 wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:8000/fastapi/sync_cpu_bound/
Running 15s test @ http://127.0.0.1:8000/fastapi/sync_cpu_bound/
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     4.60s     2.73s   14.80s    81.56%
    Req/Sec     4.23      4.76    24.00     78.87%
  Latency Distribution
     50%    4.08s
     75%    4.77s
     90%    7.37s
     99%   14.41s
  141 requests in 15.04s, 21.76KB read
Requests/sec:      9.38
Transfer/sec:      1.45KB
```

```
~/Desktop/practice/fastAPI
 wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:8000/fastapi/async_cpu_bound/
Running 15s test @ http://127.0.0.1:8000/fastapi/async_cpu_bound/
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.86s   251.07ms   2.52s    90.11%
    Req/Sec    13.86      7.65    50.00     81.96%
  Latency Distribution
     50%    1.91s
     75%    1.95s
     90%    1.99s
     99%    2.34s
  758 requests in 15.09s, 131.76KB read
Requests/sec:     50.24
Transfer/sec:      8.73KB
```

```
~/Desktop/practice/fastAPI
 wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:8000/fastapi/async_cpu_bound_slow/
Running 15s test @ http://127.0.0.1:8000/fastapi/async_cpu_bound_slow/
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     2.20s     2.20s   14.30s    92.67%
    Req/Sec     2.75      2.16     9.00     73.68%
  Latency Distribution
     50%    1.84s
     75%    2.09s
     90%    2.28s
     99%   13.07s
  150 requests in 15.05s, 26.37KB read
Requests/sec:      9.97
Transfer/sec:      1.75KB
```

Different types of API calls using Flask

Code & Explanation

```python
@app.route("/flask/sync_cpu_bound/")
def sync_cpu_bound():
    BIGNUM = 5000000
    result = executor.submit(cpu_bound_operation, BIGNUM).result()
    return {"data": result, "message": "sync mode cpu operation"}
```

This situation highlights FastAPI's ability to efficiently handle tasks by placing asynchronous functions within async path declarations. FastAPI excels by minimizing context switching. Using `asyncio.sleep(1)` for non-blocking delays showcases the framework's asynchronous capabilities, where task switching is efficiently done within a single thread. This approach avoids the overhead associated with managing multiple OS-level threads, such as thread creation and context switching costs. As a result, FastAPI offers superior execution efficiency by optimizing the use of resources and streamlining task management through asynchronous programming.

Performance Benchmarking

```
~/Desktop/practice/fastAPI
wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:5000/flask/sync_cpu_bound/
Running 15s test @ http://127.0.0.1:5000/flask/sync_cpu_bound/
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.83s   322.25ms   2.10s    91.13%
    Req/Sec    14.11      8.57    59.00     79.16%
  Latency Distribution
     50%    1.92s
     75%    1.97s
     90%    2.00s
     99%    2.07s
  767 requests in 15.10s, 163.29KB read
Requests/sec:     50.79
Transfer/sec:     10.81KB
```

Code & Explanation

```python
@app.route("/flask/sync_cpu_bound_slow/")
def sync_cpu_bound_slow():
    BIGNUM = 5000000
    cpu_bound_operation(BIGNUM)
    return {"message": "sync mode operation but pretend to run in async mode"}
```

FastAPI automatically runs synchronous functions, like `time.sleep(1)`, in a thread pool via Starlette, avoiding main event loop blockages. This means all synchronous code, regardless of its potential to block, is executed in a thread pool when called from asynchronous routes, ensuring the async event loop runs efficiently. While this automatic process by FastAPI is more efficient than manually managing thread pools in async functions, it remains subject to the constraints of synchronous operations and thread pool capacity.

Performance Benchmarking

```
~/Desktop/practice/fastAPI
wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:5000/flask/sync_cpu_bound_slow/
Running 15s test @ http://127.0.0.1:5000/flask/sync_cpu_bound_slow/
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     6.74s     3.14s   10.30s    70.20%
    Req/Sec     3.39      3.37    19.00     81.34%
  Latency Distribution
     50%    7.65s
     75%    9.65s
     90%    9.80s
     99%   10.11s
  151 requests in 15.03s, 34.21KB read
Requests/sec:     10.04
Transfer/sec:      2.28KB
```

Different types of API calls using Golang

<table>
<tr><th>Code & Explanation</th><th>Performance Benchmarking</th></tr>
<tr><td><pre><code>func asyncSleepInThreadHandler(c *gin.Context) {
    go func() {
        time.Sleep(1 * time.Second)
    }()
    c.JSON(http.StatusOK, Message{"sleep run in thread pool"})
}</code></pre></td><td><pre><code>~/Desktop/practice/fastAPI
 wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:8000/go/async_sleep_in_thread/
Running 15s test @ http://127.0.0.1:8000/go/async_sleep_in_thread/
  4 threads and 100 connections
  Thread Stats   Avg   Stdev   Max   +/- Stdev
    Latency   1.54ms   1.94ms 44.97ms   94.34%
    Req/Sec   19.75k   4.40k   66.46k   82.03%
  Latency Distribution
     50%   1.34ms
     75%   1.63ms
     90%   2.79ms
     99%   9.68ms
  1181815 requests in 15.10s, 181.46MB read
Requests/sec: 78261.29
Transfer/sec: 12.02MB</code></pre></td></tr>
<tr><td><pre><code>func syncSleepHandler(c *gin.Context) {
    time.Sleep(1 * time.Second)
    c.JSON(http.StatusOK, Message{"sync, but run in thread pool"})
}</code></pre></td><td><pre><code>~/Desktop/practice/fastAPI
 wrk -t4 -c100 -d15s --timeout 100 --latency http://127.0.0.1:8000/go/sync/
Running 15s test @ http://127.0.0.1:8000/go/sync/
  4 threads and 100 connections
  Thread Stats   Avg   Stdev   Max   +/- Stdev
    Latency   1.00s   2.14ms   1.01s   83.68%
    Req/Sec   27.52   18.12   118.00   92.19%
  Latency Distribution
     50%   1.00s
     75%   1.00s
     90%   1.01s
     99%   1.01s
  1471 requests in 15.04s, 237.03KB read
Requests/sec: 97.80
Transfer/sec: 15.76KB</code></pre></td></tr>
</table>

Benchmarking Results

Endpoint	Language/ Framwork	Requests/ sec (RPS)	Transfer (KB)/ sec	Thread Req/ Sec (Avg)	Thread Latency (s)(Avg)	Max Latency (s)	Notes
/fastapi/async_slowest/	FastAPI (Python)	0.80	0.13	0.00	6.04	12.07	Sync in async (I/O)
/fastapi/async_sleep_in_thread/	FastAPI (Python)	11.17	1.78	3.09	6.17	9.04	Sync with threads in async (I/O)
/fastapi/async_sleep/	FastAPI (Python)	93.1	14.09	56.17	1.01	1.02	Async in async (I/O)
/fastapi/sync/	FastAPI (Python)	37.24	0.13	26.53	2.38	3.03	Sync in sync (I/O)
/fastapi/async_cpu_bound/	FastAPI (Python)	50.24	8.73	13.86	1.86	2.52	Async in async (CPU)
/fastapi/sync_cpu_bound/	FastAPI (Python)	9.38	1.45	4.23	4.60	14.80	Sync in sync (CPU)
/fastapi/async_cpu_bound_slow/	FastAPI (Python)	9.97	1.75	2.75	2.20	14.30	Sync in Async (CPU)
/flask/sync_cpu_bound/	Flask (Python)	50.79	10.81	14.11	1.83	2.10	Sync with processes in sync (CPU)
/flask/sync_cpu_bound_slow/	Flask (Python)	10.04	2.28	3.39	6.74	10.30	Sync in sync (CPU)
/go/sync/	Go	97.80	15.76	27.52	1.00	1.01	
/go/async_sleep_in_thread/	Go	78261.29	12.02MB	19.75k	1.54ms	44.97ms	