1. Aim / Purpose of the Experiment

The main goal of these experiments is to assess the performance of a custom-built HTTP server (using CivetWeb as the base) under different configurations of thread pools and concurrent load levels. The focus is on measuring server throughput (requests per second) and response time when serving specific endpoints. This evaluation helps to identify how well the server scales under load and where performance bottlenecks may occur.

2. Setup and Execution Details

Environment and Configuration:

- **Server**: Custom-built HTTP server in C based on the CivetWeb library.
- **Test Environment**: Single-core Linux machine.
- **Endpoints Tested**: `/arithmetic/fibonacci`, `/arithmetic/square`, `/arithmetic/cube`.
- **Independent Parameters**:
    - **Concurrency Level**: The number of simultaneous requests.
    - **Thread Pool Size**: The number of threads available to handle concurrent requests.
- **Load Testing Tools**:
    - **Apache Benchmark (ab)** for generating load and collecting throughput and response time data.
- **Metrics Collected**:
    - Average response time (in milliseconds).
    - Throughput (requests per second).

3. Hypothesis / Expectation

We hypothesize:

- **Throughput** will initially increase with rising concurrency but will reach a plateau as the server becomes saturated.
- **Response Time** will increase steadily with higher concurrency due to resource contention and queuing delays.
- Larger thread pools will enable better handling of concurrent requests up to a point, after which additional threads may not yield performance gains due to CPU limitations.

4. Observations from Data / Plots

4.1 Fibonacci Endpoint Analysis

Thread Pool Size 1:

- Throughput started at **5877.86 requests/sec** with 10 concurrent requests and peaked at **5971.58 requests/sec** with 20 concurrency.

- As concurrency increased to 80, throughput declined to **4968.2 requests/sec**, showing a significant decrease, suggesting the limitations of a single-thread pool in handling high concurrency.

Thread Pool Size 6:

- Higher starting throughput of **6937.7 requests/sec** at 10 concurrency, maintaining performance at **6895.6 requests/sec** with 20 concurrency.
- Throughput began to decline at 40 concurrency (**6347.19 requests/sec**) and reached **6011.42 requests/sec** at 80 concurrency, indicating better handling than a single thread but still limited by CPU.

Thread Pool Size 12:

- Starting at **6230.14 requests/sec** at 10 concurrency and peaking at **6427.15 requests/sec** at 20 concurrency.
- Declined to **6074.6 requests/sec** at 40 concurrency and **5365.38 requests/sec** at 80 concurrency, demonstrating some CPU saturation with increased thread switching.

Conclusion: For the `/fibonacci` endpoint, a thread pool size of 6 provided the most consistent and efficient handling of increasing load, balancing throughput effectively before performance began to decline.

4.2 Square Endpoint Analysis

Thread Pool Size 1:

- Started with **6601.53 requests/sec** at 10 concurrency, with a gradual decrease to **5125.84 requests/sec** at 80 concurrency.
- The pattern indicates that a single-thread pool struggles as concurrency increases, with throughput dropping due to limited concurrency handling capacity.

Thread Pool Size 6:

- Started strong at **7155.64 requests/sec** at 10 concurrency, peaking at **6708.71 requests/sec** at 40 concurrency.
- Throughput remained relatively high at **6731.74 requests/sec** at 80 concurrency, showcasing better load management compared to a single thread.

Thread Pool Size 12:

- Initially at **6557.38 requests/sec** at 10 concurrency and stayed stable up to 40 concurrency at **6432.11 requests/sec**.
- Slight decrease to **5768.34 requests/sec** at 80 concurrency, indicating more stable but limited gains over 6 threads.

Conclusion: The `/square` endpoint performed best with a thread pool size of 6, maintaining high throughput at increasing concurrency levels. Larger pools did not yield significant benefits for this moderately intensive function.

4.3 Cube Endpoint Analysis

Thread Pool Size 1:

- Initial throughput was **6681.37 requests/sec** at 10 concurrency, peaking at **6969.61 requests/sec** at 20 concurrency.
- Declined to **5210.23 requests/sec** at 80 concurrency, indicating limits in handling higher loads.

Thread Pool Size 6:

- Throughput at **6719.53 requests/sec** at 10 concurrency, peaking at **6724.5 requests/sec** at 40 concurrency.
- Performance dropped slightly to **5885.47 requests/sec** at 80 concurrency, showing reasonable load handling.

Thread Pool Size 12:

- Started at **6459.11 requests/sec** at 10 concurrency, declining to **6230.92 requests/sec** at 40 concurrency.
- Ended at **5250.72 requests/sec** at 80 concurrency, showing some improvement over a single thread but no clear gains over 6 threads.

Conclusion: The `/cube` endpoint, like the `/square`, had the best performance with a pool size of 6, which provided a balance between thread handling and CPU availability.

5. Explanation of Behavior and Inferences

Throughput Analysis

Throughput generally increased with concurrency levels up to a saturation point, after which additional concurrency did not lead to better performance due to server limits. For computationally light functions, this saturation occurred at higher concurrency levels, whereas heavier functions like `/fibonacci` hit their limit sooner.

Response Time Behaviour

Response times increased with concurrency, as expected. CPU-bound functions like `/fibonacci` had more significant increases in response time due to limited processing resources. The presence of more threads allowed the server to handle concurrent requests better, but too many threads led to context switching overhead.

Thread Pool Impact

Larger thread pools (6 or 12) enhanced the server's ability to handle concurrent requests up to a point. The optimal thread pool size depended on the function being tested. For more intensive operations, a medium-sized pool (e.g., 6 threads) performed best.

Overall Inferences

The experiments confirmed the hypothesis that throughput initially increases with concurrency before plateauing. Thread pool size significantly impacts server performance, with an optimal size balancing CPU availability and concurrency.