Efficient Big Integer Arithmetic and Matrix Multiplication Performance Analysis

Introduction

This report presents the implementation of a custom BigInt library and its application in matrix multiplication. The primary objective was to evaluate the efficiency of our BigInt implementation against the well-optimized GMP library. Benchmarking results are analyzed to highlight performance differences.

Approach

BigInt Implementation

- **Chunked Representation:** Numbers are stored in chunks of base 10^9 to optimize arithmetic operations.
- **Efficient Arithmetic Operations:** Addition, subtraction, and multiplication are implemented with attention to performance and accuracy.
- **Toom-Cook Multiplication:** Used for large numbers to improve multiplication efficiency beyond schoolbook methods.
- **Optimized Base Handling:** Multiplication by powers of base accelerates calculations.

Matrix Multiplication

- **Parallel Computation:** OpenMP is used to parallelize computations for increased efficiency.
- **Tiling Strategy:** Tiling is implemented to optimize cache usage and minimize memory latency.
- **GMP Comparison:** GMP's mpz_class is used for benchmarking against our custom BigInt implementation.

Performance Optimization

- **Parallelized Execution:** OpenMP is employed for multi-threaded processing.
- **Efficient Memory Management:** Reduced redundant memory allocations.
- **Optimized Integer Arithmetic:** Custom BigInt arithmetic is carefully optimized for better performance.

Benchmarking Results

Observations

From the collected data, GMP significantly outperforms our custom BigInt implementation across all test cases. As the matrix size and integer length increase, the performance gap between GMP and the custom BigInt widens. The following trends were observed:

Small Matrices (64x64):

- Custom BigInt: ~174ms (256-digit numbers) to ~9782ms (1024-digit numbers)
- GMP: ~8ms (256-digit numbers) to ~35ms (1024-digit numbers)

Medium Matrices (256x256):

- Custom BigInt: ~17.1s (256-digit) to ~684.7s (1024-digit)
- GMP: ~207ms (256-digit) to ~2.2s (1024-digit)

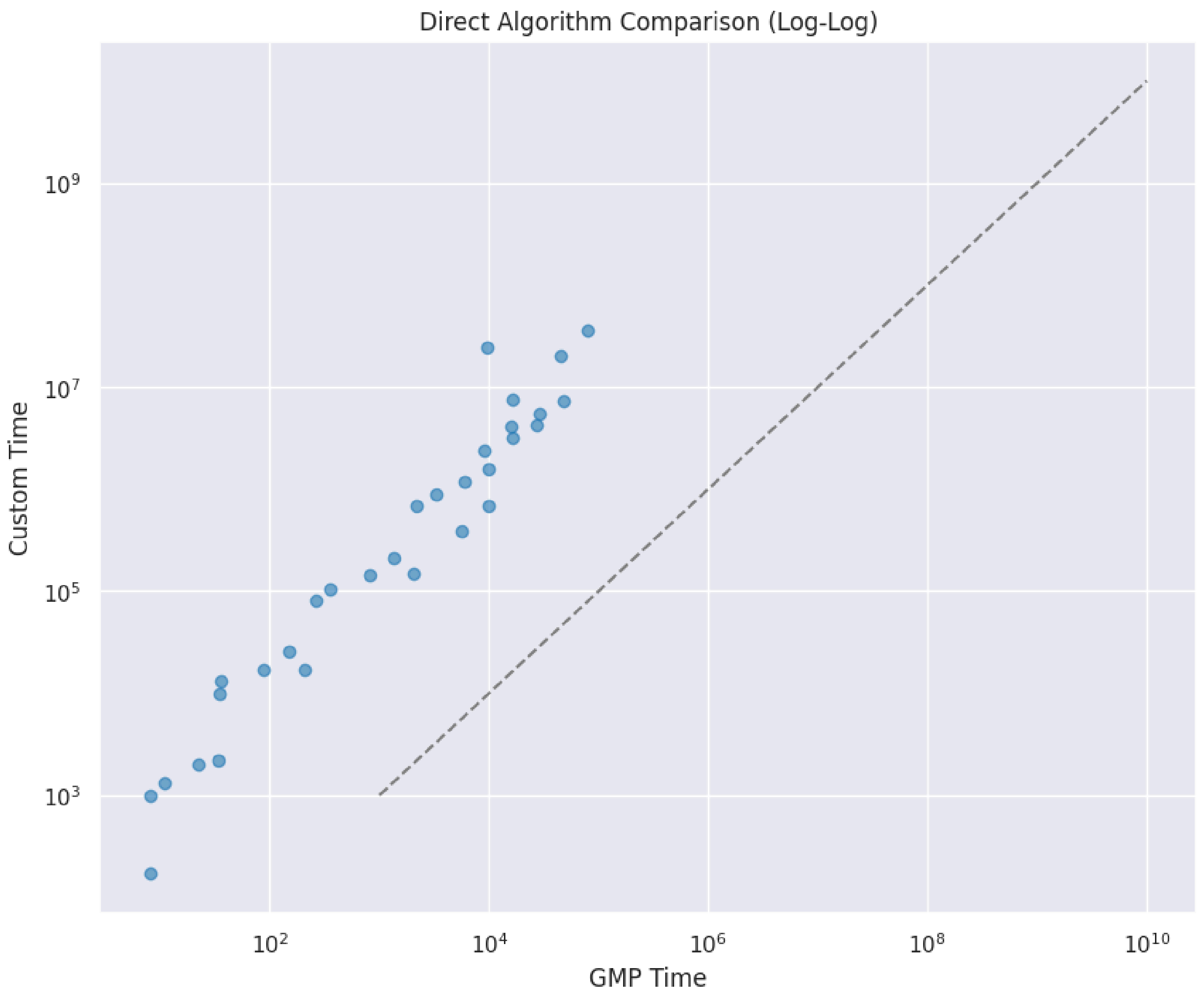
Large Matrices (1024x1024):

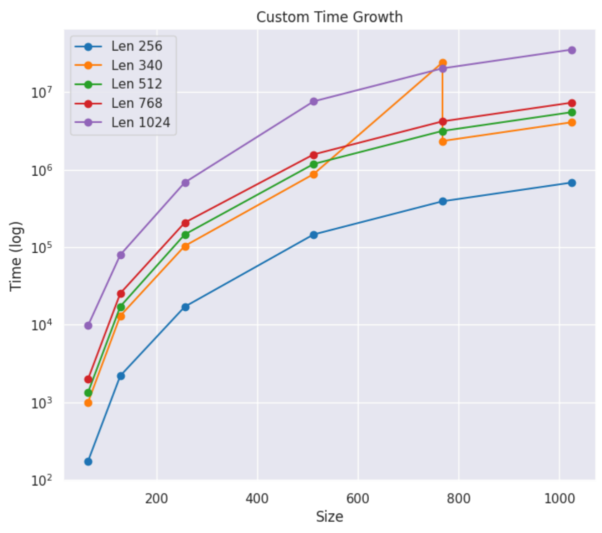
- Custom BigInt: ~682ms (256-digit) to ~35.5s (1024-digit)
- GMP: ~9.8s (256-digit) to ~78.2s (1024-digit)

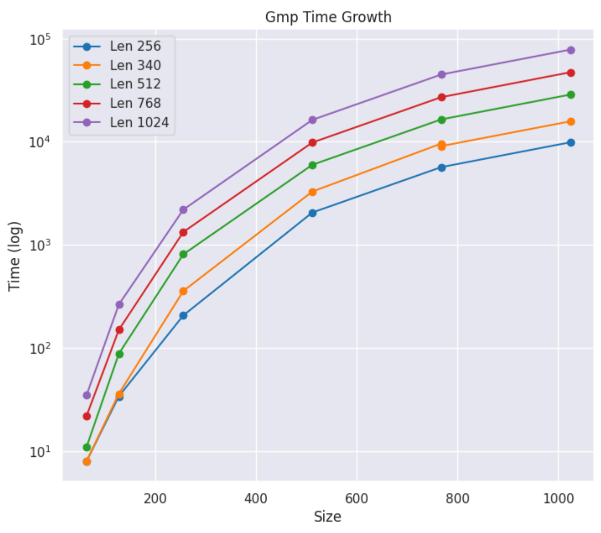
Key Findings

- **GMP is significantly faster** due to its highly optimized assembly-level implementations.
- **Custom BigInt's performance** deteriorates as the number of digits increases, highlighting inefficiencies in handling larger integers.
- **The performance gap widens significantly** as the matrix size and integer length increase, with GMP maintaining a consistent advantage.

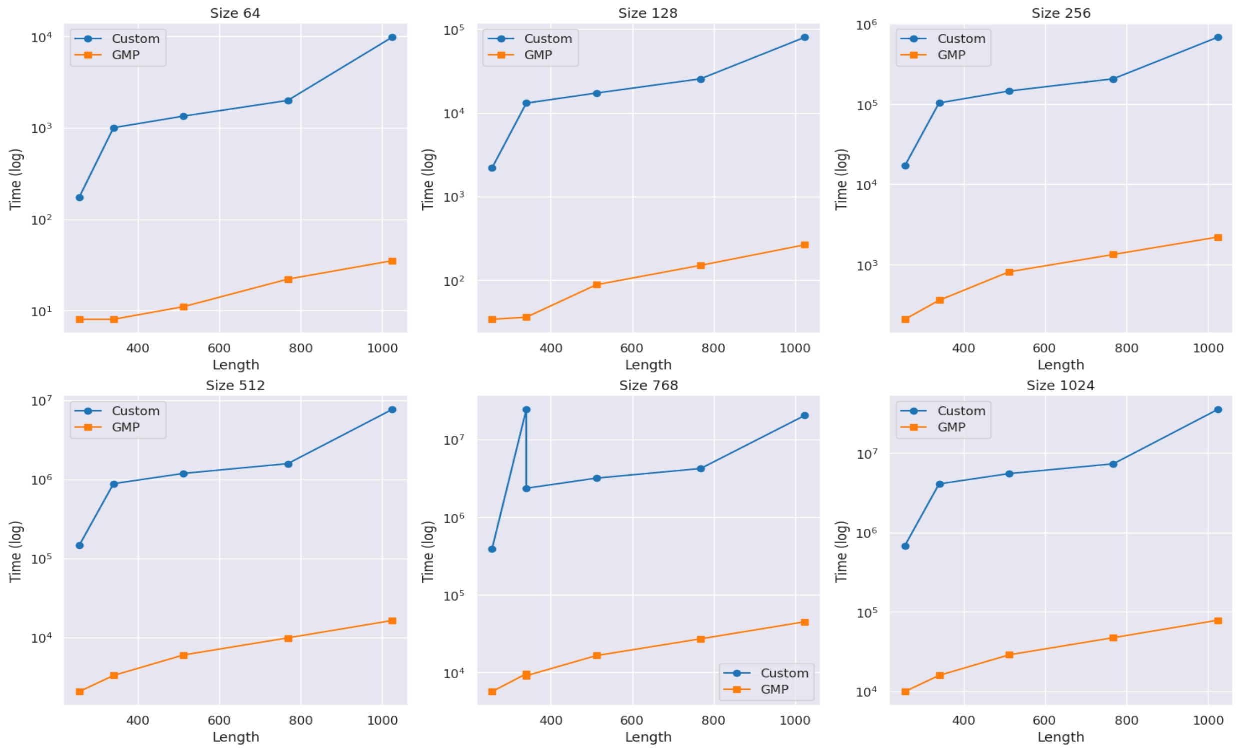
Graphs:


Direct Algorithm Comparison (Log-Log)
Custom Time
GMP Time
10^3
10^5
10^7
10^9
10^2
10^4
10^6
10^8
10^{10}

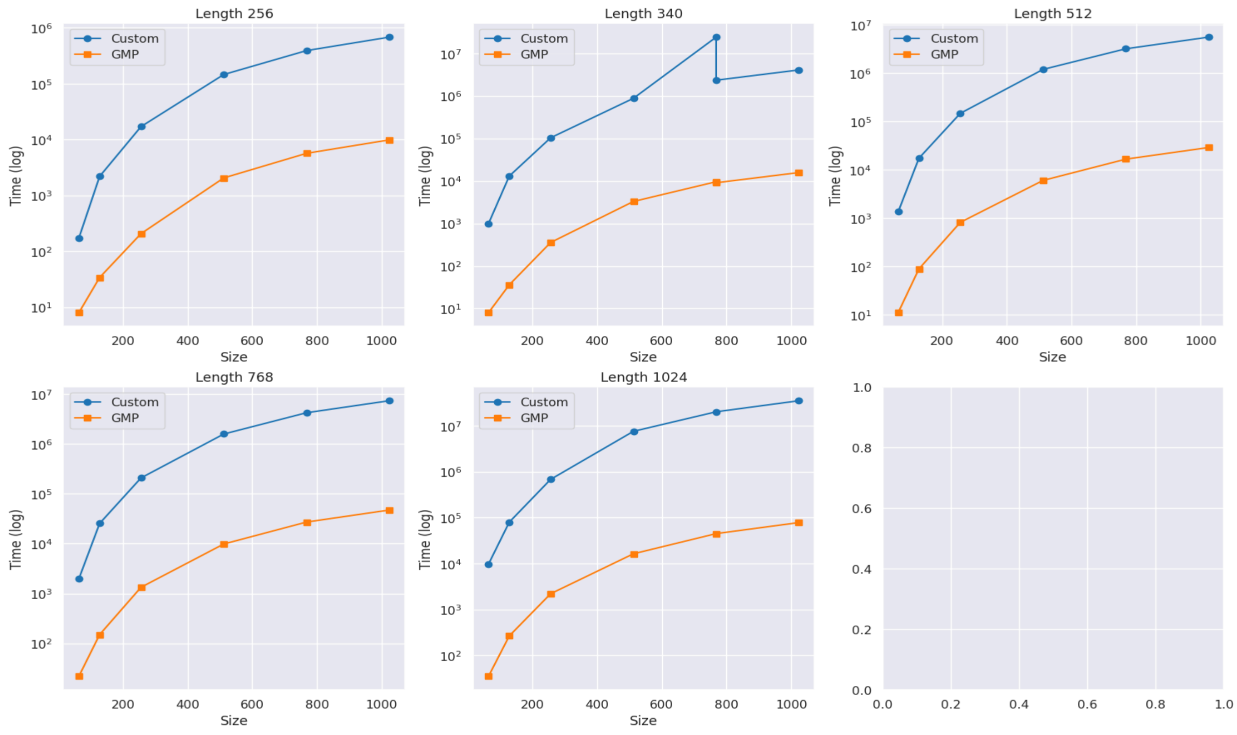

Custom Time Growth
Len 256
Len 340
Len 512
Len 768
Len 1024
Time (log)
Size
200
400
600
800
1000


Gmp Time Growth
Len 256
Len 340
Len 512
Len 768
Len 1024
Time (log)
Size
200
400
600
800
1000

Time Comparison by Length for Different Sizes (Log Scale)



Time Comparison by Size for Different Lengths (Log Scale)



Build and Execution Instructions

Compilation

Ensure you have a C++ compiler with OpenMP and GMP support.

Run:

```
$ cd /tests

$ make
```

Execution

```
$ ./test_main
```

The results will be stored in timings_output.txt. Whereas graphs are present inside /plot directory.

Conclusion

The data clearly demonstrates that GMP's optimized algorithms and assembly-level implementations provide a significant performance advantage over the custom BigInt implementation, especially as the problem size scales. The custom BigInt's performance degradation with increasing digit length and matrix size suggests a need for further optimization or a reevaluation of the underlying algorithms. GMP's consistent performance across all test cases makes it a superior choice for handling large matrices and high-digit numbers.