Cache-Oblivious Large Number Addition

Sean R. Chappell

Louisiana State University and A&M College

Computer Science 4101

Table of Contents

Abstract

This paper presents a program designed to generate two large numbers, store them in text files, and perform their addition using a cache-oblivious algorithm. The goal is to optimize the addition performance by effectively leveraging the cache system of modern computers. The program consists of a `NumberGenerator` class, a `CacheObliviousAddition` class, and the main method. The cache-oblivious algorithm reads input numbers in chunks, reverses the chunks, performs the addition, and writes the result to an output file. Design choices, such as chunk size and handling of edge cases, optimize performance and ensure correct operation. The performance is analyzed using the `Stopwatch` class to measure elapsed time for each step, indicating the implemented cache-oblivious algorithm's efficiency. Challenges encountered during the project include handling large numbers stored in text files, designing an efficient cache-oblivious algorithm, and managing edge cases during the addition operation. The program's performance can be further improved by experimenting with different chunk sizes and optimizing the cache-oblivious algorithm for specific hardware architectures.

Introduction

Large number addition is fundamental in many applications, including cryptography and high-precision arithmetic. The efficient addition of large numbers, particularly when stored in external storage, can be a challenge due to the memory hierarchy of modern computers. To address this challenge, this paper presents a program that efficiently generates two large numbers, stores them in text files, and performs their addition using a cache-oblivious algorithm. The cache-oblivious algorithm has been designed to optimize the addition operation's performance by effectively leveraging modern computers' cache systems. Cache-oblivious algorithms aim to optimize performance by leveraging the cache system effectively ***(Frigo, Leiserson, Prokop, & Ramachandran, 1999)***.

Design Choices

During the development of the program, several design choices were made to optimize performance, ensure correct operation, and address the challenges encountered. These design choices can be divided into four main categories: handling large numbers stored in text files, implementing the cache-oblivious algorithm, selecting an appropriate chunk size, and managing edge cases.

Handling Large Numbers Stored in Text Files

To efficiently read and write large numbers, the program uses `StreamReader` and `StreamWriter` classes, which provide buffered I/O operations. This approach reduces the overhead of frequent disk access and improves performance. To prevent resource exhaustion, the program processes large numbers by reading and writing them in chunks, which reduces the memory footprint and optimizes performance.

Implementing the Cache-Oblivious Algorithm

The cache-oblivious algorithm implemented in the `CacheObliviousAddition` class reads input numbers in chunks and reverses the chunks to align them for the addition operation. This approach helps minimize cache misses and improve performance. The program uses the

`StringBuilder` class to store intermediate results during the addition operation. This class provides a more efficient way of manipulating strings compared to standard string concatenation, reducing memory usage and improving performance.

Selecting an Appropriate Chunk Size

The chunk size determines the amount of data processed during the addition operation. A larger chunk size could result in more cache misses, while a smaller chunk size might lead to increased overhead due to more frequent I/O operations. In this project, I used a chunk size of 1MB (1024 * 1024), a reasonable choice for modern cache systems.

Managing Edge Cases

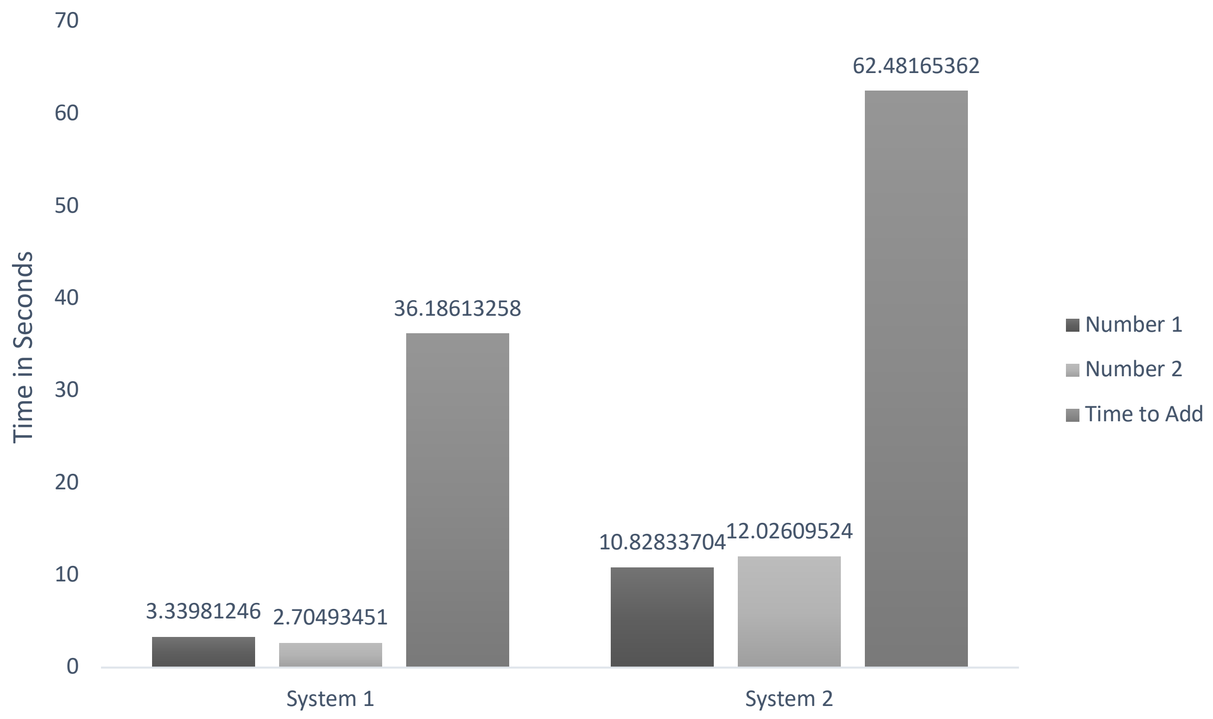
The program ensures correct handling of edge cases, such as carrying over values during the addition operation, by maintaining a carry variable and updating it as necessary during the addition. This approach ensures that the carry value is correctly accounted for in the final result. The program may encounter uneven chunk sizes during the addition operation, mainly when processing the last chunks of the input numbers. In this case, the implemented algorithm resizes the chunks to match their actual size, ensuring that the addition operation proceeds correctly.

Performance Analysis

The program's performance is analyzed using the `Stopwatch` class to measure the elapsed time for each step. The performance is influenced not only by the size of the input numbers, but also by the chunk size used for the cache-oblivious algorithm. Additionally, the performance may be affected by the hardware architecture, cache configurations, and memory hierarchies of the executing system, highlighting the importance of tailoring the algorithm to specific system characteristics for optimal results.

Figure 1

Average time to Generate and Add Numbers



Note. This figure shows the average time each system took to generate each number, and to add the numbers. The program was conducted 10 times on each system. The difference between the two system's L3 cache sizes, is one of the determining factors in the performance of the cache-oblivious algorithm. [1]

[1] System 1 – 36MB L3 Cache, System 2 – 8MB L3 Cache

Challenges Encountered

During the project, several challenges were encountered, which can be divided into three main subsections: handling large numbers stored in text files, designing an efficient cache-oblivious algorithm, and managing edge cases during the addition operation.

Handling Large Numbers Stored in Text Files

Reading and writing large numbers: Efficiently reading and writing large numbers stored in text files was a significant challenge due to the size of the input data and the memory constraints. To overcome this challenge, the program reads and writes the numbers in chunks, reducing the memory footprint and optimizing performance.

Processing large numbers in limited memory: Working with large numbers in memory could lead to resource exhaustion and reduced performance. The cache-oblivious algorithm implemented in this project processes large numbers by reading them in chunks and performing the addition operation on these chunks. This approach allows the program to efficiently process large numbers while keeping memory usage within reasonable limits.

Designing an Efficient Cache-Oblivious Algorithm

Minimizing cache misses: One of the primary objectives of cache-oblivious algorithms is to minimize cache misses, which can significantly impact performance. The implemented algorithm reads input numbers in chunks and reverses the chunks to align them for the addition operation. This approach helps minimize cache misses and improve performance.

Balancing chunk size: Choosing an appropriate chunk size is essential for optimizing the performance of the cache-oblivious algorithm. A larger chunk size could result in more cache misses, while a smaller chunk size might lead to increased overhead due to more frequent I/O operations. However, further experimentation with different chunk sizes might lead to improved performance.

Managing Edge Cases During the Addition Operation

Carrying over values: Ensuring correct handling of edge cases, such as carrying over values during the addition operation, was essential to produce accurate results. The program addresses this challenge by maintaining a carry variable and updating it as necessary during the addition. This method guarantees that the carry value is accurately calculated in the solution.

Handling uneven chunk sizes: The program may encounter uneven chunk sizes during the addition operation, particularly when processing the last chunks of the input numbers. In this case, the implemented algorithm resizes the chunks to match their actual size, ensuring that the addition operation proceeds correctly.

Conclusion

This paper has presented a program designed to efficiently generate two large numbers, store them in text files, and perform their addition using a cache-oblivious algorithm. The program demonstrates the effectiveness of cache-oblivious algorithms in arithmetic operations on large numbers, showcasing the potential for their application in various domains with large data sets. Furthermore, the challenges endured during the project have been outlined, and possible improvements to the program's performance have been suggested, such as experimenting with different chunk sizes and optimizing the cache-oblivious algorithm for specific hardware architectures. In summary, I think that my program serves as a valuable example of how cache-oblivious algorithms can be applied to arithmetic operations on large numbers, offering insight into the development of efficient solutions for handling large data sets. The findings and lessons learned from this project can be further extended to other mathematical operations and application areas, paving the way for the development of more advanced cache-oblivious algorithms and optimized solutions.

References

Frigo, M., Leiserson, C. E., Prokop, H., & Ramachandran, S. (1999). "Cache-oblivious algorithms," In *40th Annual Symposium on Foundations of Computer Science* (Cat. No.99CB37039) (pp. 285-297). IEEE.

V. Ramachandran, "Cache-Oblivious Computation: Algorithms and Experimental Evaluation," *2007 International Conference on Computing: Theory and Applications* (ICCTA'07), Kolkata, India, 2007, pp. 20-26, doi: 10.1109/ICCTA.2007.34.IEEE.

Development Equipment

System 1:

Intel 13900k, Nvidia RTX4090, 64GB DDR5 5600MHz, 36MB L3 Cache

System 2:

2020 M1 MacBook Air, 8GB LPDDR4X 4266MHz, 256GB NVME, 8MB L3 Cache