Running Speed of Fibonacci Number Algorithms

Name: Haoran Miao

Student Number: 301542189

SFU email: [hma98@sfu.ca](mailto:hma98@sfu.ca)

**Introduction :**

The objective of this project is to explore and analyze four different algorithms for generating Fibonacci numbers, which were discovered on the https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/ Fibonacci numbers are a sequence of integers in which each number is the sum of the two preceding ones, starting from 0 and 1. The 4 algorithms under consideration are Space Optimized Method, Recursive Approach, Dynamic Programming, Nth Power of Matrix Approach.

Each of these algorithms provides a unique approach to compute Fibonacci numbers, and their performance can vary significantly depending on factors such as code optimization, instruction utilization, memory locality, and branch predictability. In this project, we will aim to improve the running speed of the Fibonacci number generation by evaluating and optimizing these algorithms.

Fibonacci using Space Optimized Method: Result: 102334155 Time taken: 0.000000151 seconds.

Fibonacci using Recursion: Result: 102334155 Time taken: 0.164332179 seconds.

Fibonacci using Dynamic Programming: Result: 102334155 Time taken: 0.000000080 seconds.

Fibonacci using Nth Power of Matrix Approach: Result: 102334155Time taken: 0.000000153 seconds.

This is the test result of finding the 40th Fibonacci number. From the above result we can conclude that the recursive method is the slowest, the Dynamic programming is the fastest. Which was not what I predicted; I thought the Nth power Matrix method was going to be the fastest.

int fib\_dynamic(int n){

int f[n + 2];

int i;

f[0] = 0;

f[1] = 1;

for (i = 2; i <= n; i++) {

        f[i] = f[i - 1] + f[i - 2];

}

  return f[n];

}

This is the code for the dynamic method, it has a running time of O(n). In this function, the cache was not used well, when dealing with big number of n, the running speed will be slower. The branch predictions was very clear and simple, and is only executed at the start of the function.

**Speed Modify Attempt:**

int fib\_dynamic\_cache(int n){

if (n <= 0) return 0;

if (n == 1) return 1;

int prev\_prev = 0; // The (n-2)th Fibonacci number

    int prev = 1;      // The (n-1)th Fibonacci number

    int result = 0;    // The nth Fibonacci number

    for (int i = 2; i <= n; i++) {

        result = prev + prev\_prev;

prev\_prev = prev;

prev = result;}

return result;

}

This modified function has a faster running speed of 0.000000061 seconds. Which is faster than the version before. In this function, I deleted the use of arrays, and switch to storing the values in variables, this caused the function to access less memory than the one before, this function is also storing only 3 variables prev, prev\_prev, and result, which means it uses less memory space, so it can run in the high level cache, making the code run faster. The branch predictions is one extra than the previous code, it check if n <=0 or n ==1, but won’t affect the over all speed of the code.

Fib\_dynamic (40) = 102334155, Cycles taken: 184

Fib\_dynamic\_cache (40) = 102334155, Cycles taken: 109

By looking at the CPU cycles, we can see that the modified version takes almost 50% less CPU cycles than the unchanged version.

**Assembly Convert**:

gcc -std=c17 -O2 -march=haswell -S fibonacci.c -o This is the linux command I used to convert the functions into assembly code.

I deleted some of the machine code, kept the assembly code, and did some compare with the two versions. The unchanged version of fib\_dynamic function contains a very long code, with 8 different functions that it uses to jump around, the whole code was very unclear, and contains to many unused memory and registers. The modified version of the function, is very short, contains only a loop, and a base case return. I will be doing some modifying on top of the fib\_dynamic\_cache function, and compare the speed with c functions.

The complier did a great job in writing the assembly code, I did not find any better ways to improve the core loop of the function in assembly. But the start of the function where it checks if n <=0 or n= 1, the complier used 2 compare instructions, test and cmp. My modified version used only test, this made the whole running speed faster, and I was able to beat the original code.

Compiler’s function, compared with modified function.

A screenshot of a computer screen

Description automatically generatedA screenshot of a computer program

Description automatically generatedA screenshot of a computer program

Description automatically generated

The above result used fib (46), the largest int where there is no overflow, and took the average of 50 tests. The modified function using assembly is not faster than the C function, but it beats the compiler generated assembly code, and uses the least cycles. On the other hand, by comparing the modified C functions and the modified assembly code with the original Algorithm, the modified all have better performance than the original function.

**Disadvantage:**

The old Dynamic Fibonacci function stores the whole Fibonacci sequence, and the new modified function only contain the Nth Fibonacci value. Both of the functions used int for the header, meaning the function can only calculate maximum of 46th Fibonacci value, if a greater n is passed in, it will create a overflow. Because the Dynamic Fibonacci function uses array, the function can not take of negative int value, but the new modified version can.

**Conclusion:**

This project explored and analyzed four different algorithms for generating Fibonacci numbers: Space Optimized Method, Recursive Approach, Dynamic Programming, and Nth Power of Matrix Approach. I did not get the functions from Rosetta Code, because some of the functions did not generate the correct value, and one function was not able to run.

The test results showed that the recursive method was the slowest, while the dynamic programming approach was the fastest. To further improve the dynamic programming algorithm, a modified version was implemented, which utilized variables instead of arrays, leading to a significant reduction in memory access and faster execution. Additionally, the modified C function outperformed the compiler-generated assembly code, demonstrating the effectiveness of the optimization. Overall, the modified version has defeated the original function in both running speed and CPU cycles.

**Reference:**

<https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/>

(used for providing the methods for Fibonacci)