Parallelizing and Optimizing the Needleman-Wunsch Algorithm

Project - Software Programming for Performance, Monsoon 2022

Arjun Muraleedharan (2020101099)
Keyur Chaudhari (2020101100)

Problem Statement

Needleman-Wunsch algorithm is an algorithm that is used in bioinformatics to align DNA or protein sequences. It involves the use of dynamic programming to compare biological sequences. The Needleman-Wunsch algorithm is used to find the optimal global alignment, and essentially divides a larger problem into a series of smaller problems to find this optimal alignment.

In our project, we attempt to make improvements upon the baseline performance of the algorithm by parallelizing it and optimizing it to achieve the peak performance possible.

Compute Configuration

- Model: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz(140)
- Generation: 11th
- Frequency Range: 400 MHz-4200 MHz
- Number of Cores: 8
- Hyper Threading Availability: Yes (threads/core = 2)
- Cache Size: L1d cache - 192 KiB, L1i cache - 128 KiB, L2 cache - 5 MiB, L3 cache - 8 MiB
- Peak FLOPS per core = 33.6 GFLOPS
- Main Memory Size: 8 GB
- RAM Type: DDR4
- Main Memory Bandwidth (obtained using the Stream Benchmark Program): 18730.15 MB/s

Correctness Evaluation Approach

Our first priority, before comparing the optimization parameters, was to test the correctness of each optimization.

In order to test the correctness of our program, we first wrote the baseline program and ensured its correctness via manual checking and computation of the result. Moreover, we tried testing for large strings and verified with versions that were known to be accurate that our results were accurate as well.

Once we had ensured that our baseline program was correct, the correctness of all subsequent optimizations were verified via comparison with the output of the baseline program.

Performance Evaluation Framework

In order to measure the performance of our optimizations, we had to settle upon metrics to be used to determine improvement.

The main performance metric we used was **execution time**. We were able to judge the level of optimization in terms of speed that the changes added to the baseline program using this parameter.

A secondary, albeit important parameter we used was **throughput**. The amount of throughput delivered by the optimized versions was something we looked at as well.

Datasets Used

1. We started off with pairs of predetermined strings of small lengths like 5 and 4, in order to verify the correctness of our programs.
2. We then moved on to larger randomized strings as a way of giving a proper measure of the performance of the program and its performance relative to the other optimized versions of it.
3. Finally, we tested our programs using the sample datasets provided by the TAs, provided in the FASTA format, comparing output with standard implementations to verify accuracy.

Baseline Performance

The baseline performance of the original, unoptimized Needleman-Wunsch program on the compute system is as follows. The program was run on strings of length 10,000.

- **Execution Time:** 2777.16 ms
- **Operational Intensity:** 1.5
- **Throughput:** 0.216048 GIPS

This was the case when no parallelization, compiler optimization or locality exploitation was added to the code, and the program involved the brute-force version of the algorithm.

Optimization Techniques Tried

Technique 1: Exploiting Cache Locality

We tried a variety of techniques, attempting to optimize the program and obtain the best possible performance from it. The very first strategy that came to mind was **spatial cache locality**. Since memory accesses are significantly slower than cache accesses, our goal was to maximize the number of cache hits.

We tried experimenting with the program to test this out. In the computation of the DP matrix, we switched the order of the nested loops, such that we iterated through the matrix column-wise, and measured the performance.

We then switched the order of the nested loops again, such that the matrix was iterated row-wise, and measured the performance again.

Technique 1: Exploiting Cache Locality

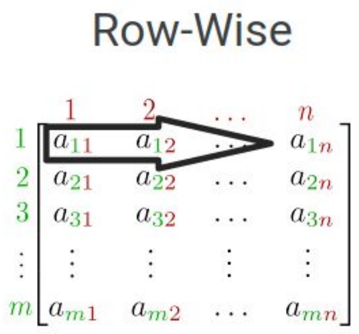
Indeed, cache locality ended up speeding up the program by a factor of **1.35**. This was our first major optimization. The matrix is stored in row-major form on the system, and so accessing row-wise is a good strategy in terms of spatial locality.

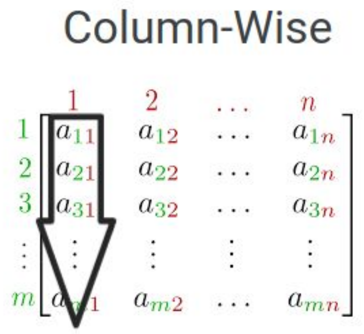
Column-Wise:

- **Execution Time:** 2777.16 ms
- **Operational Intensity:** 1.5
- **Throughput:** 0.216048 GIPS

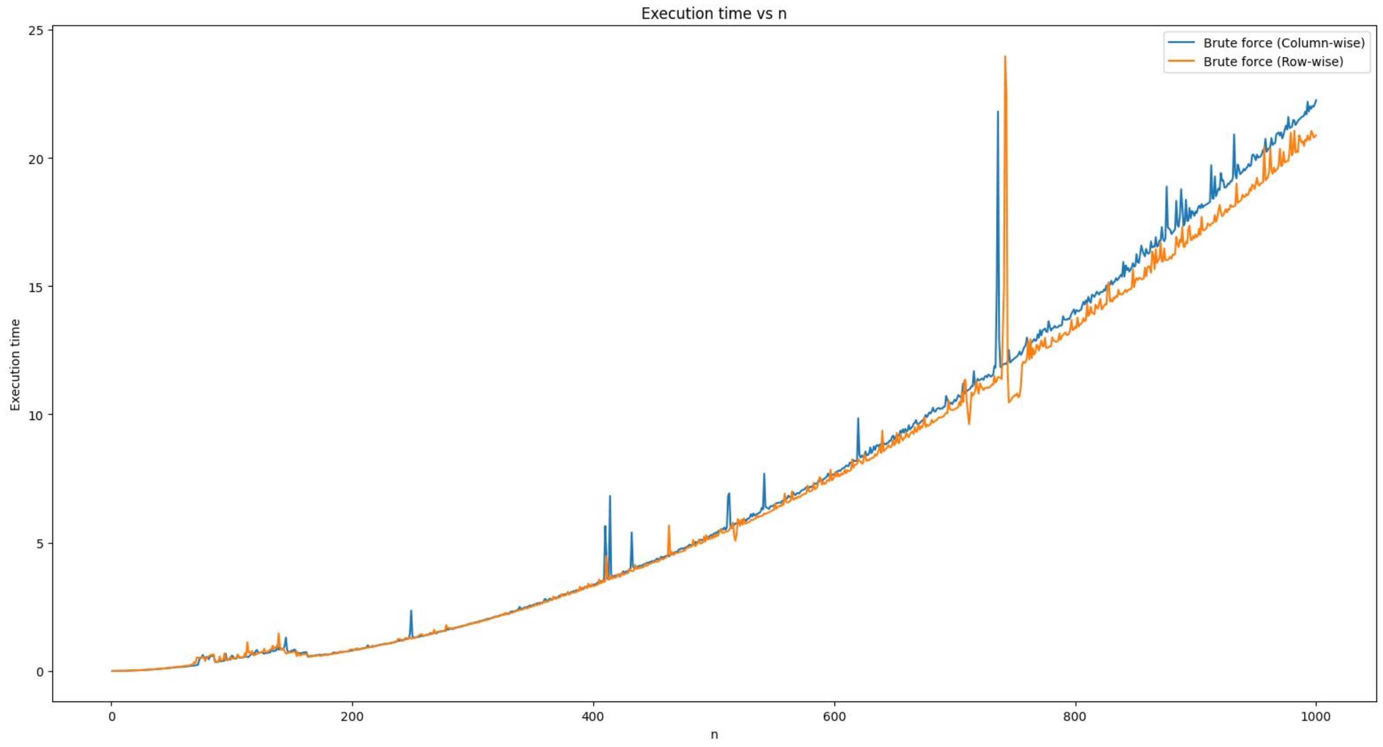
Row-Wise:

- **Execution Time:** 2056.67 ms
- **Operational Intensity:** 1.5
- **Throughput:** 0.291733 GIPS





Performance Analysis of Row-Wise vs Column-Wise



Inferrals

Overall, exploiting cache locality did improve performance, but not as much as we thought it would.

Ultimately, the program had no parallelization, and so the performance we obtained was not going to be the best until we added parallelization.

Still, this was progress, and exploiting locality gave us a performance optimization.

Technique 2: Compiler Optimizations

In order to extract as much performance from the program as possible, we added compiler flags (like -O3) in order to optimize the compiled program as much as possible. In addition, we also used unsuppressed optimizations by adding the following pragma directives at the beginning of our program.

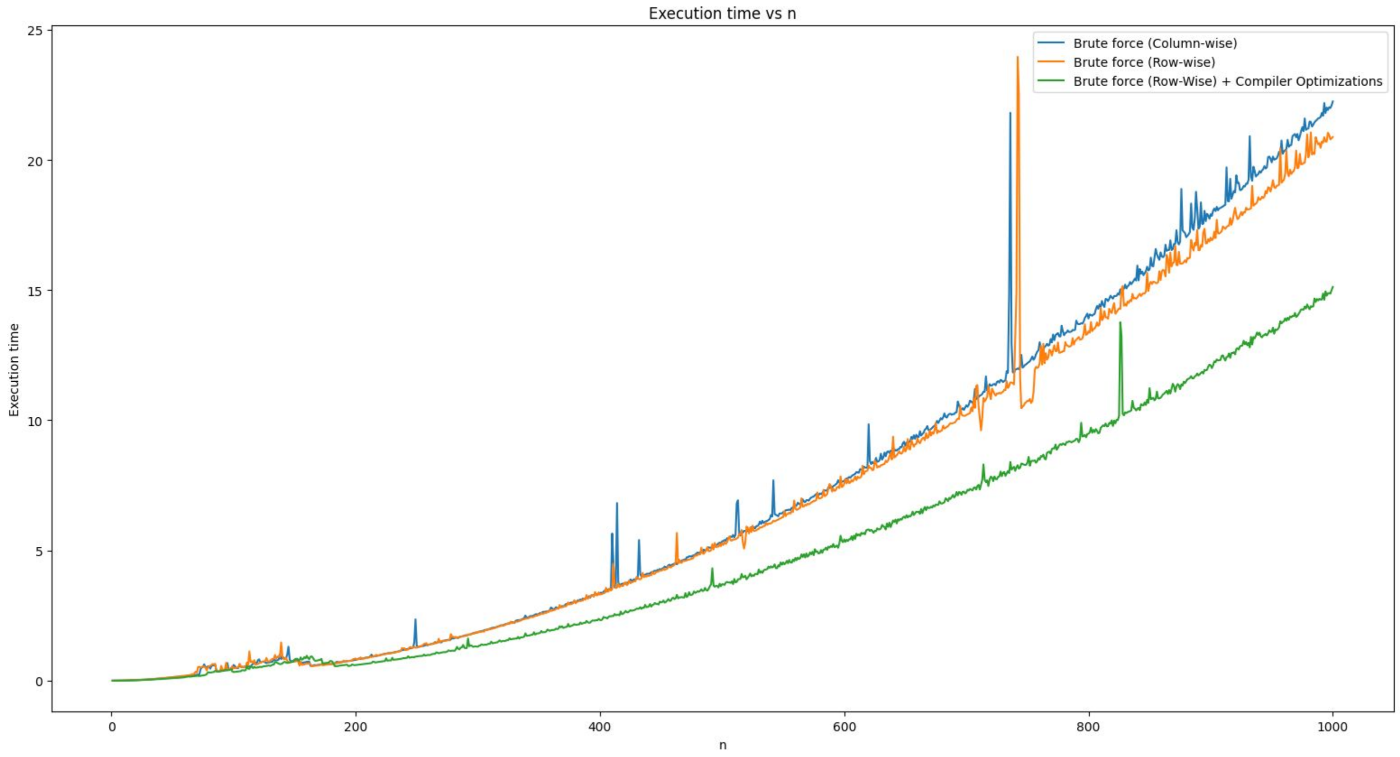
```
#pragma GCC optimize(“O3”)

#pragma GCC optimize(“Ofast”)
```

In addition, we switched to the ICC compiler from GCC. All these changes sped up our program by approximately **1.3**.

- **Execution Time:** 1587.8 ms
- **Operational Intensity:** 1.5
- **Throughput:** 0.37788 GIPS

Performance Analysis with and without Compiler Optimizations



Inferrals

Adding the pragmas and switching to the ICC compiler gave us a drastic increase, as shown in the plot.

This shows that even though we can add optimizations within the program code, we are sometimes restricted by the higher level of the programming language and the compiler can still perform several optimizations under the hood to extract as much performance from the program as possible.

Another observation was that on this particular system (which has an Intel Core i5 processor), the ICC compiler gives better results than GCC.

Technique 3: Parallelization along the Anti-Diagonal

Even though exploiting spatial locality by ensuring we access the DP matrix row-wise, it wasn't enough, since we were not **parallelizing** our program. Our next goal was to add parallelization to the program.

However, this wasn't possible with the original program. The reason is that the Needleman-Wunsch algorithm being a dynamic programming algorithm, each solution is computed based on the previous solutions (i.e. dp[i][j] depends on dp[i-1][j], dp[i-1][j-1] and dp[i][j-1]). This means that there are several dependencies across the iterations, meaning that we cannot simply parallelize our loops without there being possible inconsistencies in the matrix.

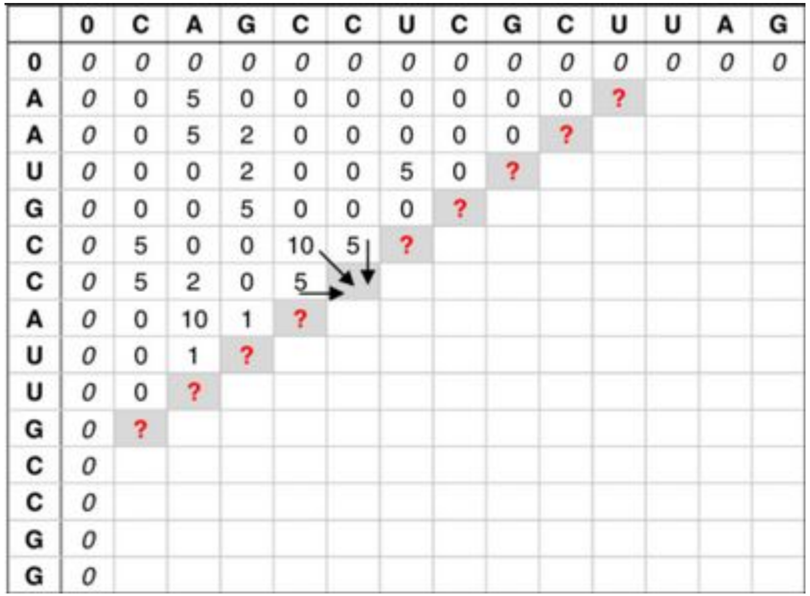
This required us to change our approach to the **anti-diagonal approach**.

Technique 3: Parallelization along the Anti-Diagonal

In the anti-diagonal approach, instead of iteration row-wise or column-wise, we iterate along the anti-diagonal, as shown in the diagram.

This works because dp[i-1][j] and dp[i][j-1] are computed in the previous anti-diagonal to dp[i][j], and dp[i-1][j-1] in the anti-diagonal before that. Therefore, there is no inconsistency or conflict in data.

However, since the values in an anti-diagonal do not depend on each other in any way, we can use parallel threads to compute the values along the anti-diagonal. **Thus, we have successfully added parallelism to the program**.



	0	C	A	G	C	C	U	C	G	C	U	U	A	G
0	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*	*0*
A	*0*	0	5	0	0	0	0	0	0	0	?			
A	*0*	0	5	2	0	0	0	0	0	?				
U	*0*	0	0	2	0	0	5	0	?					
G	*0*	0	0	5	0	0	0	?						
C	*0*	5	0	0	10	5	?							
C	*0*	5	2	0	5									
A	*0*	0	10	1	?									
U	*0*	0	1	?										
U	*0*	0	?											
G	*0*	?												
C	*0*													
C	*0*													
G	*0*													
G	*0*													

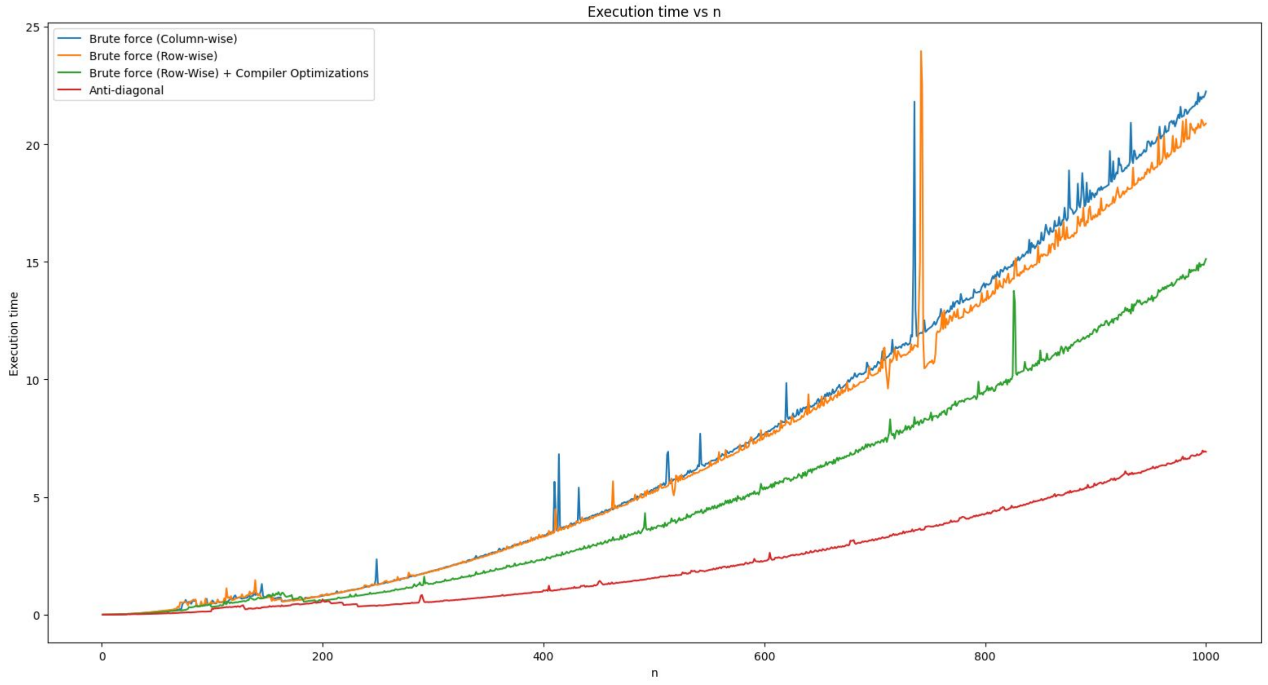
Technique 3: Parallelization along the Anti-Diagonal

Parallelization ended up speeding up our program by a lot (in fact, by a factor of **1.7** from the previous major optimization and **2.29** from the baseline program). Clearly, it was a drastic increase and shows the impact parallelism can have on program performance. This was our second major optimization.

- **Execution Time:** 1209.75 ms
- **Operational Intensity:** 1.5
- **Throughput:** 0.495971 GIPS

The anti-diagonal approach allowed us to exploit parallelism to obtain speedup. However, there was still more we could do.

Performance Analysis of Unoptimized vs. Anti-diagonal Approach



Inferrals

Clearly, parallelism gave us a huge advantage, as evidenced from the plot. With the normal, unoptimized algorithm, the execution speed increases exponentially, even with compiler optimizations (since the time complexity is O(HW) - essentially quadratic, since we took H and W to be equal for this particular analysis).

However, for the anti-diagonal approach, combined with compiler optimizations, this increase is reduced significantly as the size of the string increases, thanks to parallelization. The exponentiation of the execution time is less for the anti-diagonal approach.

Technique 4: Tiling using Submatrices

The issue with the normal anti-diagonal approach was that while it allowed us to use parallelism, it eliminated the benefit of spatial locality that row traversal gave us. We attempted to figure out a way to make use of parallelism while still retaining the benefits of cache locality.

Our approach was **tiling**. Tiling is a technique wherein the original DP matrix is divided into square submatrices. We then fill each submatrix with the corresponding values just like we do for the normal matrix, using row traversal. We proceed in this manner, filling all the submatrices and filling the leftover cells using brute force, iterating through all of them and filling them based on the values of the previous cells.

This is useful because it allows us to make use of locality (even if it sometimes may not be as much as normal row traversal).

Technique 4: Tiling using Submatrices

However, tiling on its own neither gives us an improvement, nor does it allow us to parallelize our program. It is when we combine it with the anti-diagonal approach where we start to see results.

Here, we fill each submatrix using row-wise traversal. However, **each submatrix is filled by a separate thread**. This is done by selecting the submatrices to be filled in an anti-diagonal order, similar to the original anti-diagonal approach. As the submatrices in each diagonal do not depend on each other, parallelism can be used to fill each submatrix. Therefore, not only do we make use of locality, we also make use of parallelism to fill each submatrix. This can make a huge difference when the strings are of huge lengths, and so the DP matrix is of huge dimensions.

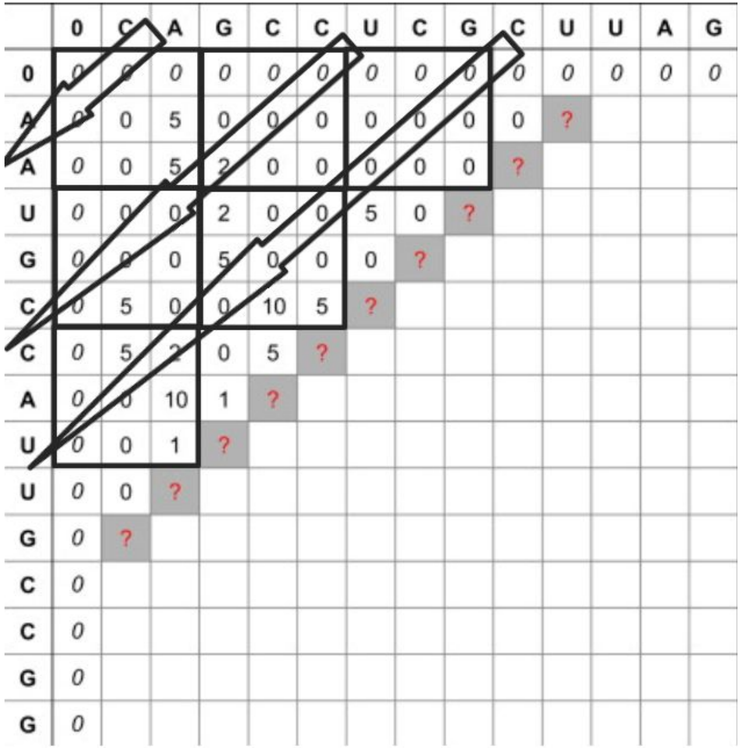
Technique 4: Tiling using Submatrices

This technique gave us the best performance of all. This is for strings of length 10000 and tile size 4.

- **Execution Time:** 1080.17 ms
- **Operational Intensity:** 1.5
- **Throughput:** 0.555469 GIPS

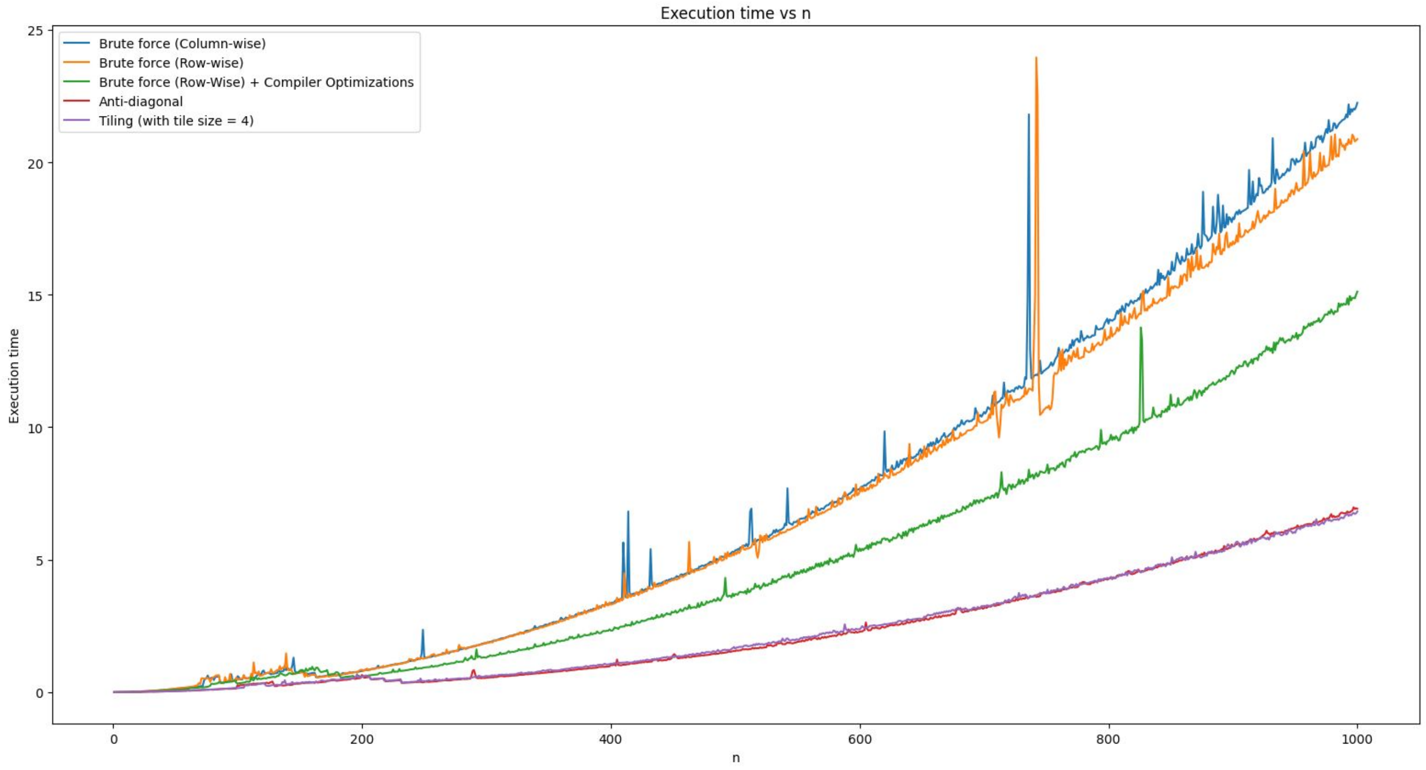
This gave us a speedup of approximately **1.2** from the anti-diagonal approach and an overall speedup of **2.57** from the baseline program.

Clearly, exploiting both parallelism and cache locality paid off well in terms of performance.

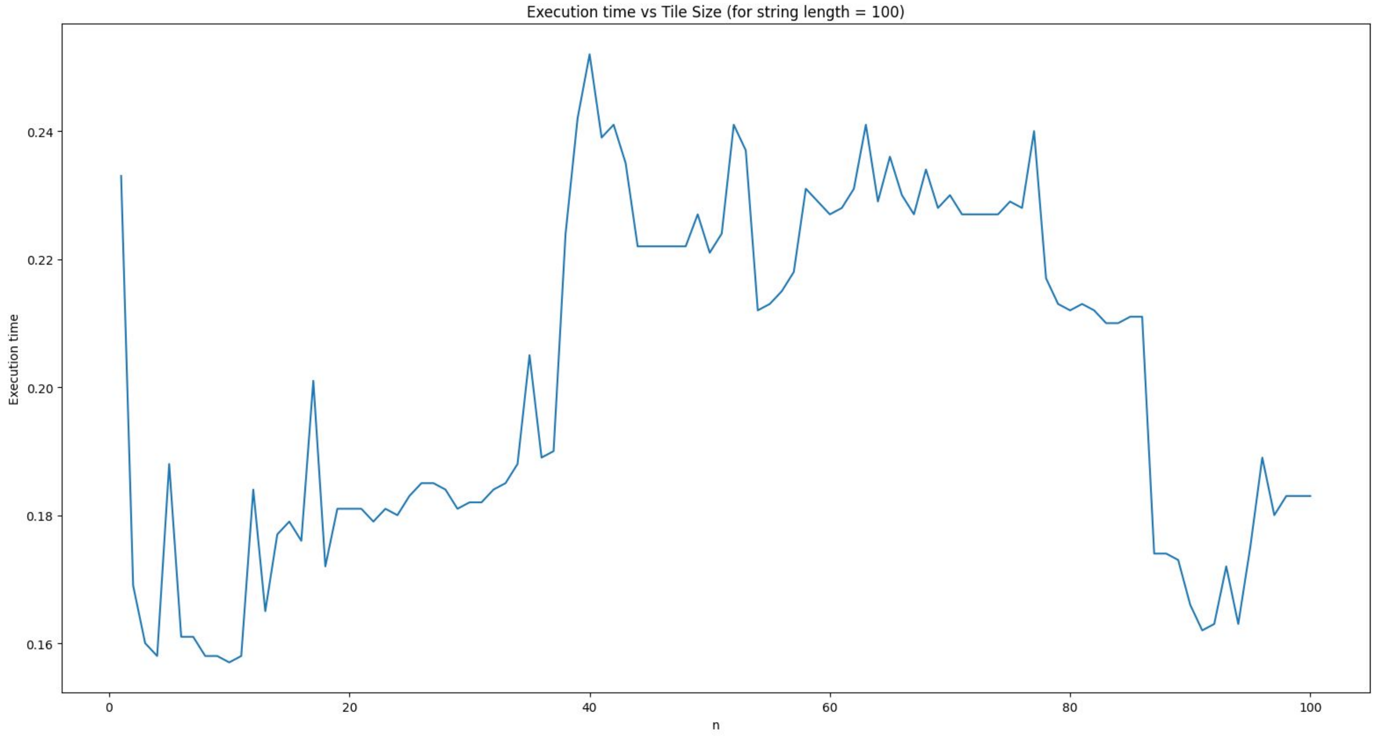


	0	C	A	G	C	C	U	C	G	C	U	U	A	G
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	5	0	0	0	0	0	0	0	?			
A	0	0	5	2	0	0	0	0	0	?				
U	0	0	0	2	0	0	5	0	?					
G	0	0	0	5	0	0	0	?						
C	0	5	0	0	10	5	?							
C	0	5	2	0	5	?								
A	0	0	10	1	?									
U	0	0	1	?										
U	0	0	?											
G	0	?												
C	0													
C	0													
G	0													
G	0													

Performance Analysis of Other Optimizations vs. Tiling



Performance Analysis of Tiling with Different Tile Sizes



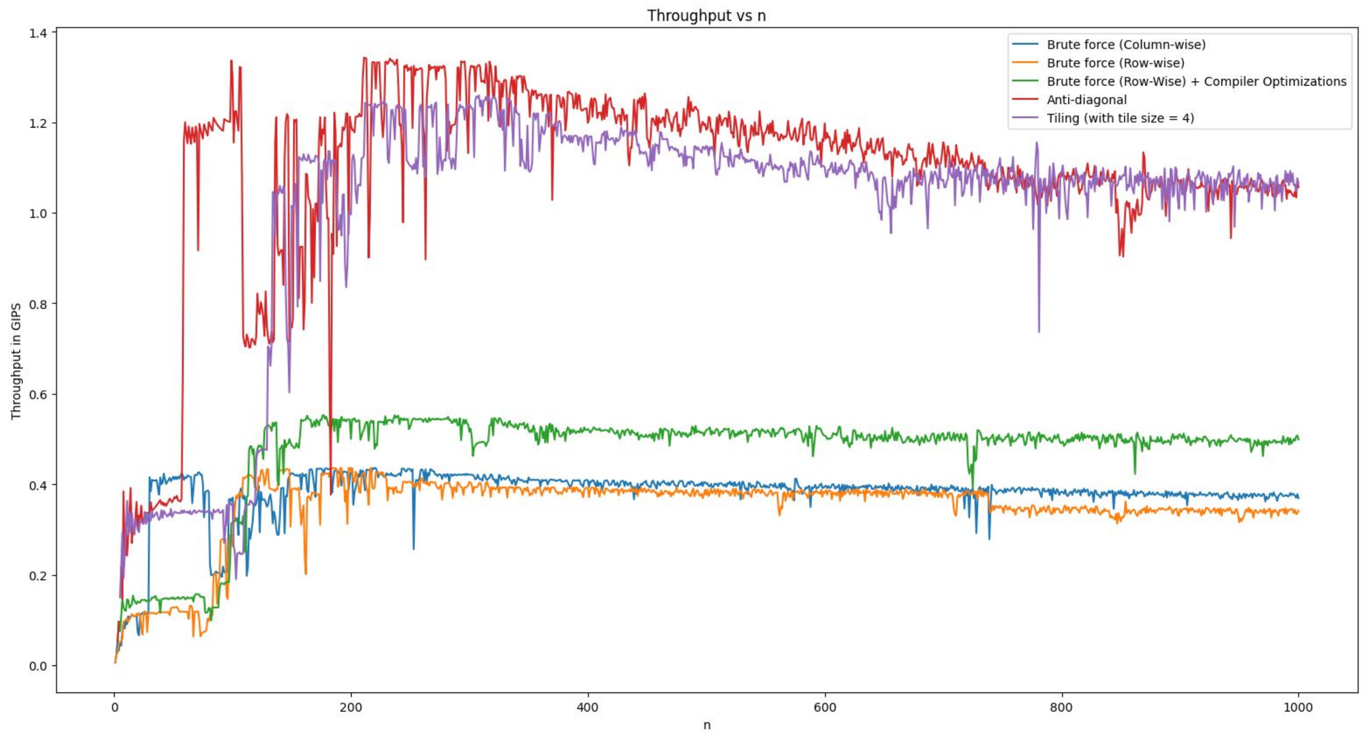
Inferrals

Since we were curious about the effect tile size would have on the execution time, we decided to try out the program for different tile sizes, for strings of length 100. Of course, the results could vary depending on string length.

We observed that for our set constraints, tile size 40 seemed to give the highest execution time, and the lowest execution time seems to be for approximately tile size 10.

Tile size can affect how cache locality is exploited by the program.

Throughput Analysis of Various Optimizations



Inferrals

Clearly, the tiling and anti-diagonal approaches give the best performance in terms of throughput. WIth each optimization, the throughput increases.

This, together with the execution time plots, indicates how parallelization, compiler optimizations and cache locality can drastically improve the performance of our program.

In fact, tiling and the anti-diagonal approach tend to give throughputs above 1 GIPS, while for brute force, it rarely goes above 0.6 GIPS.