UNIVERSITY OF SALERNO

DEPARTMENT OF INFORMATION ENGINEERING AND ELECTRICAL AND APPLIED MATHEMATICS

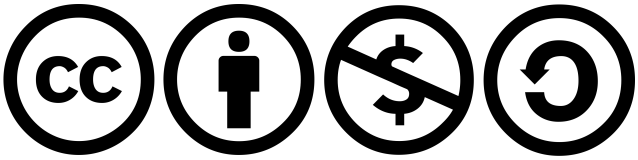
MASTER'S DEGREE IN COMPUTER ENGINEERING

High Performance Computing 2025/2026
I-Z

MANBER-MYERS SUFFIX ARRAY

Lecturer:
Ch. Prof.
Francesco Moscato
fmoscato@unisa.it

Student:
Marco Panico
Matricola n° 0622702416
m.panico20@studenti.unisa.it



Accademic Year 2025/2026

Contents

List of Figures

1 Introduction

1.1 Problem Description

A well-established approach to solving the *Longest Repeated Substring (LRS)* problem relies on the construction of **Suffix Arrays**, a space-efficient alternative to suffix trees that enables the lexicographical ordering of all suffixes of a given string. Among the algorithms designed for building suffix arrays, the **Manber–Myers algorithm** stands out for its conceptual simplicity and computational efficiency, achieving a time complexity of $O(n \log n)$. Despite this efficiency, the algorithm can still become computationally intensive when dealing with very large input strings, making it an excellent candidate for parallelization.
The objective of this project is to **develop and evaluate parallel versions** of the Manber–Myers algorithm using two complementary parallel programming paradigms:

1. **OpenMP**, which provides shared-memory parallelism on multi-core CPU architectures through compiler directives; and

2. **CUDA**, which leverages the massive data-parallel capabilities of modern GPUs to accelerate computationally intensive operations.

To ensure fair and reproducible performance comparisons, a series of **input strings of fixed memory sizes**—approximately 1 MB, 50 MB, 100 MB, 200 MB, and 500 MB—has been generated. Each of these strings is used as input for all algorithmic variants (sequential, OpenMP, and CUDA) under identical conditions.
The study aims to measure and analyze the impact of parallelization on the execution time and scalability of the algorithm across different architectures, thereby providing quantitative insights into the effectiveness of CPU-based and GPU-based parallelization strategies under controlled and consistent input conditions.

1.2 Manber–Myers Algorithm

The **Manber–Myers algorithm** is an efficient method for constructing a *Suffix Array*, a simple data structure that provides a sorted list of all suffixes of a string. Suffix arrays offer similar query performance to *Suffix Trees* but require three to five times less memory, making them more practical for large-scale text analysis and data compression.
Suffix arrays allow substring searches in $O(P + \log N)$ time, where P is the pattern length and N the text length. Although suffix trees can be built in linear time for small alphabets, the original Manber–Myers algorithm constructs suffix arrays in $O(N \log N)$, with an improved version achieving expected linear time.

Overall, the Manber–Myers algorithm provides a scalable and memory-efficient method for suffix array construction, forming the basis for many modern text indexing and pattern-matching applications.
Let us define a few key concepts before describing the algorithm:

- **Suffix:** Given a string S of length n, a suffix is any substring that starts at position i and extends to the end of the string, denoted as $S[i, n-1]$.
- **Suffix Array (SA):** An array of integers representing the starting positions of all suffixes of S in lexicographical order.
- **Rank:** A numerical value associated with each suffix, indicating its relative position in the sorted order. Initially, the rank of each suffix is determined by its first character.

The **core idea** of the Manber–Myers algorithm is to iteratively sort the suffixes based on the first h characters, where h starts at 1 and doubles at each step. Rather than comparing full suffix strings directly, each suffix is represented by a pair of integer ranks, which allows the sorting to be performed efficiently using integer operations.

1.2.1 Algorithm Description

The Manber–Myers algorithm constructs the suffix array in an iterative manner using a doubling strategy. Let n be the length of the string S, and let h be an integer that starts at 1 and doubles at each iteration.

Initialization. Each suffix is initially ranked according to its first character. The initial suffix array is obtained by sorting the indices of the suffixes based on these ranks.

Iterative Ranking. For each value of h:

- Each suffix at position i is represented by a pair of ranks: $(\text{rank}[i], \text{rank}[i+h])$, where $\text{rank}[i+h]$ is set to -1 if $i + h \geq n$.
- The suffixes are then sorted based on these rank pairs, typically using a stable sort.
- New ranks are assigned according to the lexicographical order of the pairs, ensuring that suffixes with identical pairs receive the same rank.

Termination. The process continues until $h \geq n$, at which point all suffixes are fully distinguished. The suffix array then contains the starting positions of the suffixes in lexicographical order.

Advantages. By representing suffixes with integer pairs instead of full strings, the algorithm avoids costly string comparisons, achieving a time complexity of $O(n \log n)$ and space complexity of $O(n)$. This approach also lends itself naturally to parallelization, since both rank computation and sorting steps can be efficiently distributed across multiple cores or GPU threads.

Example: Consider the string $S =$ `banana$`. Initially, the suffixes are ranked by their first character: all suffixes starting with 'a' receive the same rank, while those starting with 'b', 'n', or '$' receive distinct ranks. In each iteration, the algorithm doubles the comparison length and updates the ranks based on previously computed values, until all suffixes are uniquely ranked.

The final suffix array for S is:

$$SA = [6, 5, 3, 1, 0, 4, 2]$$

which corresponds to the lexicographically sorted order of suffixes: `$`, `a$`, `ana$`, `anana$`, `banana$`, `na$`, `nana$`.

2 Experimental setup

2.1 Experimental setup OpenMP

2.1.1 Hardware

CPU

The experiments were conducted using a Windows Subsystem for Linux (WSL) environment. The machine specifications are listed below:

Architecture: x86_64

CPU op-mode(s): 32-bit, 64-bit

Address sizes: 39 bits physical, 48 bits virtual

Byte Order: Little Endian

CPU(s): 8

On-line CPU(s) list: 0-7

Vendor ID: GenuineIntel

Model name: 11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz

CPU family: 6

Model: 140

Thread(s) per core: 2

Core(s) per socket: 4

Socket(s): 1

Stepping: 1

BogoMIPS: 4838.41

Flags:] fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc arch_perfmon rep_good nopl xtopology tsc_reliable nonstop_tsc cpuid tsc_known_freq pni pclmulqdq vmx ssse3 fma cx16 pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp ibrs_enhanced tpr_shadow ept vpid ept_ad fsgsbase tsc_adjust bmi1 avx2 smep bmi2 erms invpcid avx512f avx512dq rdseed adx smap avx512ifma clflushopt clwb avx512cd sha_ni avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves vnmi avx512vbmi umip avx512_vbmi2 gfni vaes vpclmulqdq avx512_vnni avx512_bitalg avx512_vpopcntdq rdpid movdiri movdir64b fsrm avx512_vp2intersect md_clear flush_l1d arch_capabilities

Virtualization: VT-x

Hypervisor vendor: Microsoft

Virtualization type: full

Caches (sum of all):

- L1d: 192 KiB (4 instances)
- L1i: 128 KiB (4 instances)
- L2: 5 MiB (4 instances)
- L3: 8 MiB (1 instance)

RAM

The system memory was measured using both WSL and Windows PowerShell. The combined results are listed below:

Total Memory: 8 GB

Slots used: 8 of 8

Format factor: SODIMM/DIMM

Reserved for hardware: 166 MB

Available Memory: 1.1 GB

Cache: 1 GB

Paging pool: 646 MB

Non-paging pool: 559 MB

In use (compressed): 6.6 GB (174 MB)

WSL measurement: The system memory as measured in WSL is as follows: total memory of 3.7 GiB, with 456 MiB used, 3.3 GiB free, and 3.5 MiB shared. The buffers/cache occupy 164 MiB, leaving 3.3 GiB available for applications. The swap space totals 1.0 GiB, all of which is currently free.

2.1.2 Software

The software environment used for compiling and running the project is summarized below.

Operating System: Ubuntu 24.04

Compiler: GCC 13.3.0

IDE / Editor: Visual Studio Code 1.105.1

Build System: GNU Make 4.3

Python Interpreter: Python 3.12.3

2.2 Experimental setup CUDA

2.2.1 Hardware

CPU
The experiment were conducted with the same CPU for both the parallelize version.
GPU NVIDIA

```
Mon Dec  8 14:05:16 2025
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 560.35.02              Driver Version: 560.94         CUDA Version: 12.6     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce GTX 750 Ti      On  |   00000000:01:00.0  On |                  N/A |
| 40%   0C    P8              1W /   38W |     615MiB /   2048MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A        32      G   /Xwayland                                   N/A      |
+-----------------------------------------------------------------------------------------+
```

2.2.2 Software

CUDA Version: 12.6

GCC Version: 13.3.3

NVCC Version: 12.0 (V12.0.140)

2.3 String generation

The strings were generated randomly and then saved to a text file. For all tests, the same set of strings was used. The generated strings occupy a total memory of 1 MB, 50 MB, 100 MB, 200 MB, and 500 MB. Since the total number of data structures is 8, the generated strings represent 1/8 of each structure.

3 OpenMP

3.1 Problem description

Provide a parallel version of the Member-Myers algorithm with OpenMP.

3.2 Solution

The implementation is based on the Manber–Myers algorithm, a classical $O(n \log n)$ method for suffix array construction. The sequential version is used as the baseline for the development of the parallel version.
Performance is evaluated by comparing sequential and parallel execution times, with the goal of reducing overall runtime while preserving the correctness of the suffix array.

3.2.1 Sequential

The sequential implementation of the Manber–Myers algorithm follows the classical steps:

- **Input loading:** the program reads the input text file and converts it into an integer array representing each character.
- **Initial sorting:** a counting sort is applied to the characters to initialize the suffix ordering and bucket structure.
- **Prefix-doubling loop:** the main loop iteratively doubles the prefix length h and refines bucket boundaries. Auxiliary arrays such as `pos`, `rank_arr`, `bh`, `b2h`, `cnt`, and `next_bucket` are used for efficient updates.
- **Performance measurement:** execution time is measured using CPU clocks and recorded to a CSV file, providing a baseline for parallelization.

The sequential version ensures the correctness of the suffix arrays and serves as a reference for evaluating the speedup of the OpenMP parallel implementation.

3.2.2 Parallel

Two parallel variants of the Manber–Myers algorithm were evaluated. Both originate from the same sequential implementation, but differ in the extent of loop-level parallelism.

- **First approach: full parallelization.** All `for` loops amenable to parallel execution were annotated with OpenMP directives. Particular care was required to avoid race conditions in the counting sort (array `freq`) and during bucket refinement (array `cnt`), handled through critical sections or OpenMP synchronization mechanisms.
- **Second approach: reduced parallelization.** In this version, three loops previously parallelized were intentionally reverted to sequential execution. Empirical profiling showed that, for these specific loops, the overhead of thread scheduling and synchronization exceeded the amount of useful work. As a result, parallelization degraded performance instead of improving it.

The main scalability limitations appear in the loops that iterate over individual buckets:

```
for (int j = i; j < next_bucket[i]; j++)
```

As the algorithm progresses and h doubles, most buckets eventually shrink to a single element, so the number of iterations of these loops becomes very small. In such conditions, the cost of parallel overhead dominates.
Below are the three performance-critical examples, which were parallelized in the first approach and kept sequential in the second.

1) Rank initialization

```
for (int i = 0; i < n; i = next_bucket[i]) {
    cnt[i] = 0;
    for (int j = i; j < next_bucket[i]; j++)
        rank_arr[pos[j]] = i;
}
```

When buckets contain only a few elements, each iteration performs a negligible amount of work. Parallelizing such a loop results in scheduling overhead far greater than the computation itself.

2) Bucket refinement

```
for (int j = i; j < next_bucket[i]; j++) {
    int s = pos[j] - h;
    if (s >= 0) {
        int head = rank_arr[s];
```

```
        rank_arr[s] = head + cnt[head]++;
        b2h[rank_arr[s]] = 1;
    }
}
```

This loop requires synchronization on shared structures such as `cnt` and `b2h`. When only a few iterations occur per bucket, the cost of critical sections or atomic operations outweighs any benefit from parallel execution.

3) Bucket boundary cleanup

```
for (int j = i; j < next_bucket[i]; j++) {
    int s = pos[j] - h;
    if (s >= 0 && b2h[rank_arr[s]]) {
        for (int k = rank_arr[s] + 1;
             k < n && !bh[k] && b2h[k];
             k++)
            b2h[k] = 0;
    }
}
```

This loop includes inter-iteration dependencies and a nested forward scan. These structural properties severely limit parallel efficiency. In practice, the parallel version was consistently slower.

Rationale for Removing Parallelism in These Loops At the beginning of the algorithm, buckets contain many suffixes (for large strings), so parallel execution is beneficial. As h increases, buckets gradually split until each contains a single element, so that `next_bucket[i]` $\approx$ `i+1`. Moreover, the number of buckets approaches n (`for (int i = 0; i < n; i = next_bucket[i])`), and within each iteration a parallel loop over `for (int j = i; j < next_bucket[i]; j++)` executes very few iterations, creating significant overhead. In this regime, parallelization is inefficient due to:

- negligible useful work per iteration,
- thread creation and scheduling dominating runtime,
- synchronization costs becoming significant,
- an increasing sequential fraction limiting speedup according to Amdahl's Law.

$$S = \frac{1}{(1-P) + \frac{P}{N}}$$

When the parallelizable portion P becomes too small, increasing the number of threads does not improve performance and may even slow down execution. For

these reasons, the second parallel version avoids parallelizing the three loops shown above, achieving better overall efficiency despite containing less parallel code.

3.3 Analysis

The performance evaluation was conducted by measuring the execution time, speed-up, and efficiency of the Manber-Myers algorithm. All tests were performed on input strings of increasing size: 1 MB, 50 MB, 100 MB, 200 MB, and 500 MB.
Parallel tests were executed using OpenMP with 1, 2, 4, and 8 threads.
The performance analysis was structured as two case studies, corresponding to the two parallelization strategies:

- **Case Study 1: Full parallelization** — tested with all five input string sizes.
- **Case Study 2: Reduced parallelization** — tested with all five input string sizes.

For each case study, the following metrics were recorded:

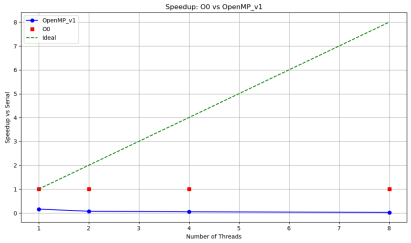
- **Execution time:** wall-clock time required to construct the suffix array.
- **Speed-up:** ratio of sequential execution time to parallel execution time for each number of threads.
- **Efficiency:** speed-up divided by the number of threads, providing a measure of parallel utilization.

This setup allows a clear comparison of how each parallelization strategy performs across different input sizes and varying numbers of threads, highlighting the effect of input size on parallel scalability.

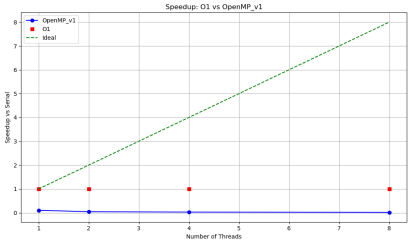
3.4 Case Study 1

Below are the results of the sequential version and version 1 of OpenMP for all the input strings.

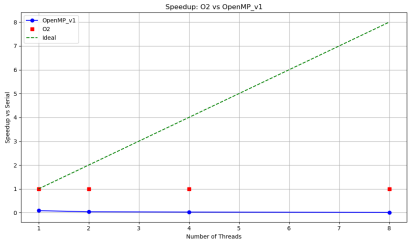
String 1MB



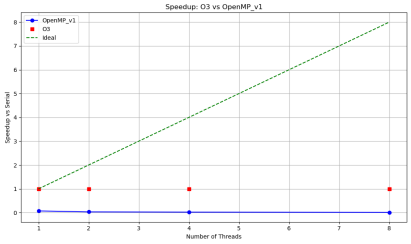
(a) Speedup with no optimization



(b) Speedup with O1 optimization



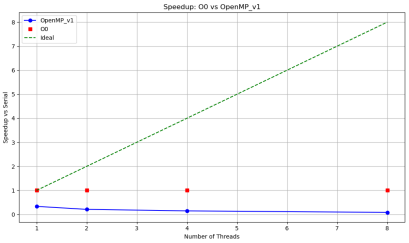
(c) Speedup with O2 optimization



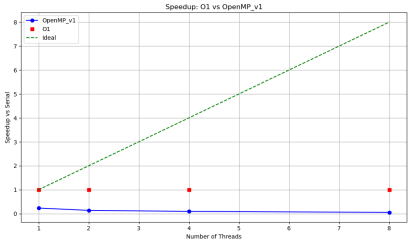
(d) Speedup with O3 optimization

Figure 1: Speedup for input 1MB

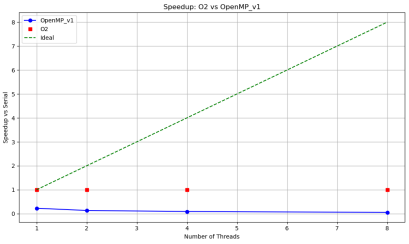
String 50MB



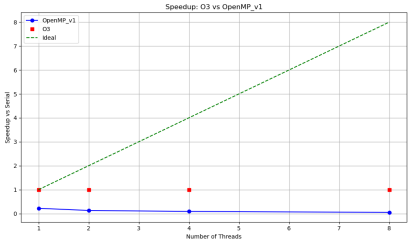
(a) Speedup with no optimization



(b) Speedup with O1 optimization



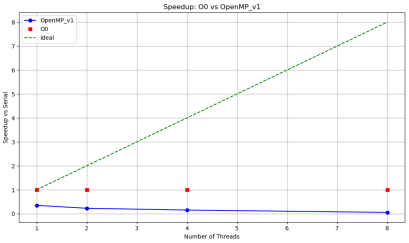
(c) Speedup with O2 optimization



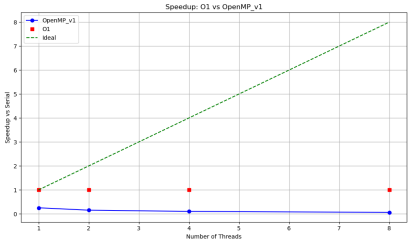
(d) Speedup with O3 optimization

Figure 2: Speedup for input 50MB

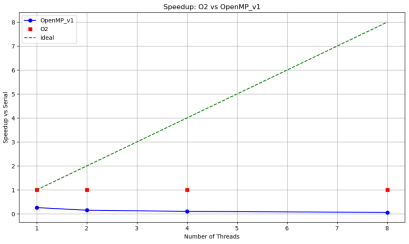
String 100MB



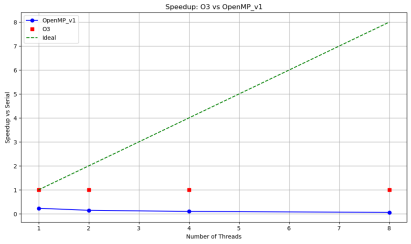
(a) Speedup with no optimization



(b) Speedup with O1 optimization



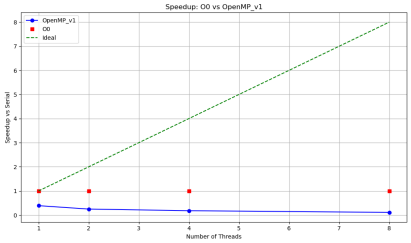
(c) Speedup with O2 optimization



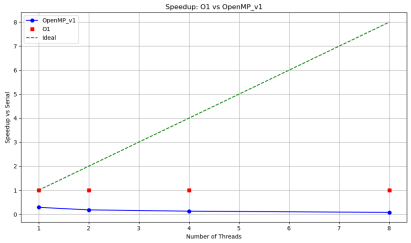
(d) Speedup with O3 optimization

Figure 3: Speedup for input 100MB

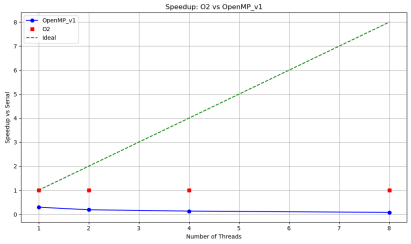
String 2000MB



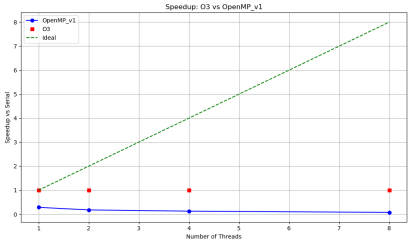
(a) Speedup with no optimization



(b) Speedup with O1 optimization



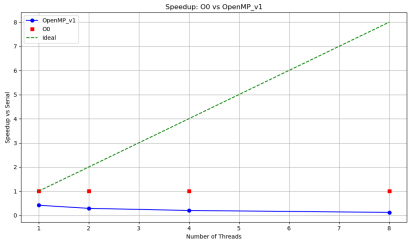
(c) Speedup with O2 optimization



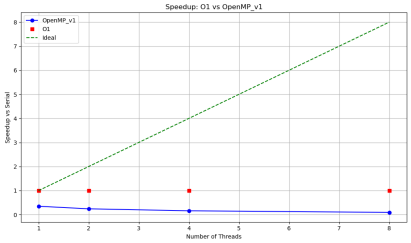
(d) Speedup with O3 optimization

Figure 4: Speedup for input 200MB

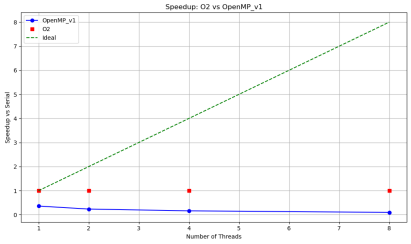
String 500MB



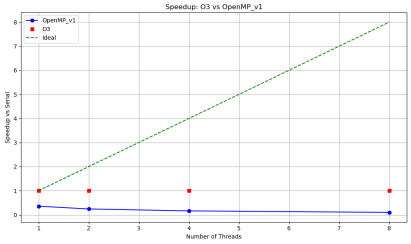
(a) Speedup with no optimization



(b) Speedup with O1 optimization



(c) Speedup with O2 optimization



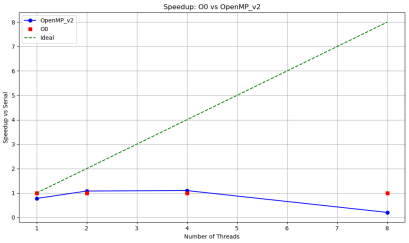
(d) Speedup with O3 optimization

Figure 5: Speedup for input 500MB

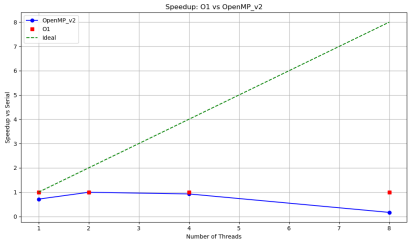
3.5 Case Study 2

Below are the results of the sequential version and version 2 of OpenMP for all the input strings.

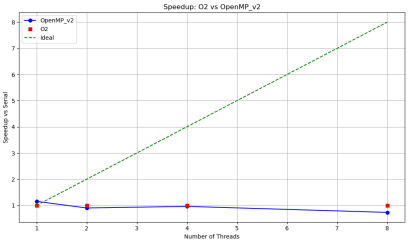
String 1MB



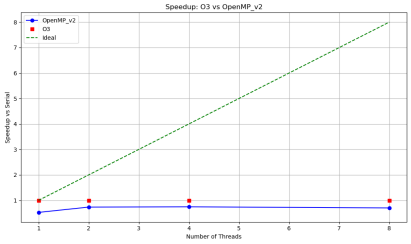
(a) Speedup with no optimization



(b) Speedup with O1 optimization



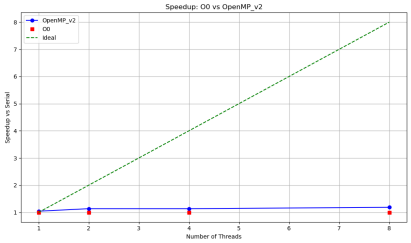
(c) Speedup with O2 optimization



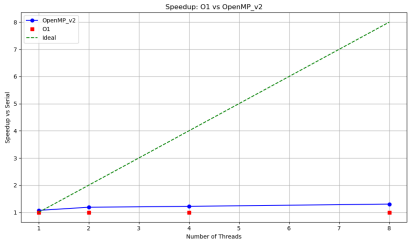
(d) Speedup with O3 optimization

Figure 6: Speedup for input 1MB

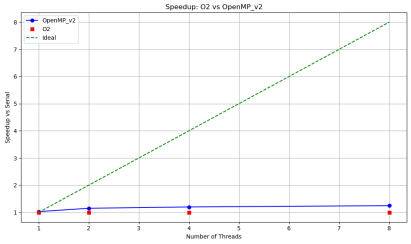
String 50MB



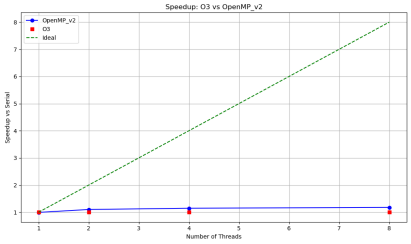
(a) Speedup with no optimization



(b) Speedup with O1 optimization



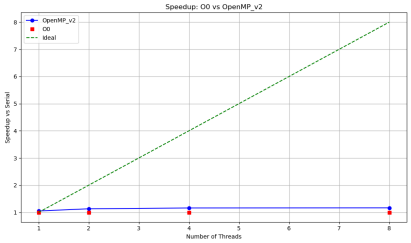
(c) Speedup with O2 optimization



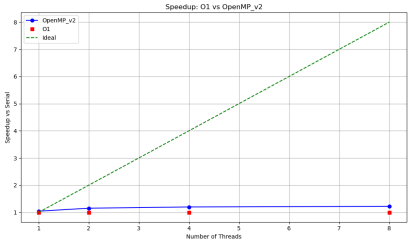
(d) Speedup with O3 optimization

Figure 7: Speedup for input 50MB

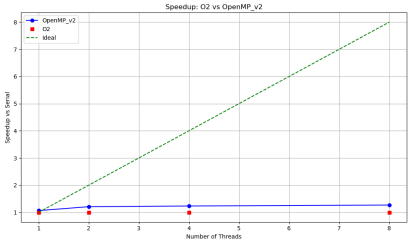
String 100MB



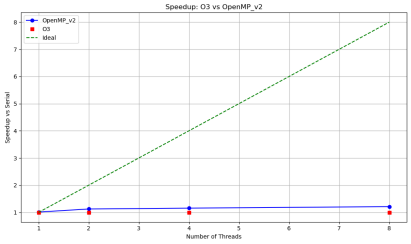
(a) Speedup with no optimization



(b) Speedup with O1 optimization



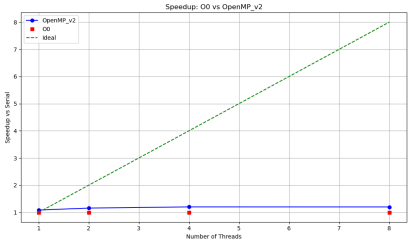
(c) Speedup with O2 optimization



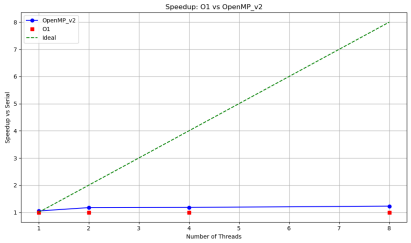
(d) Speedup with O3 optimization

Figure 8: Speedup for input 100MB

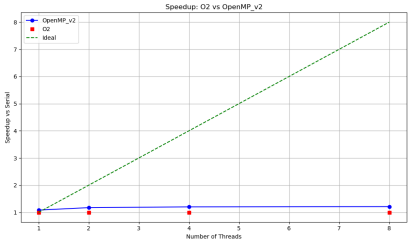
String 2000MB



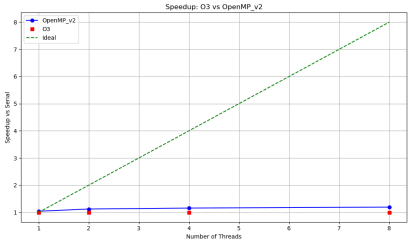
(a) Speedup with no optimization



(b) Speedup with O1 optimization



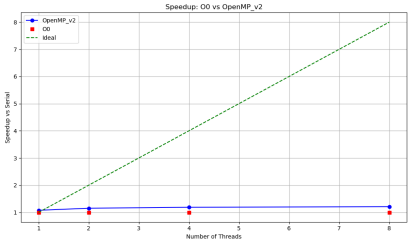
(c) Speedup with O2 optimization



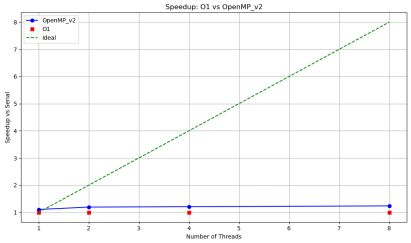
(d) Speedup with O3 optimization

Figure 9: Speedup for input 200MB

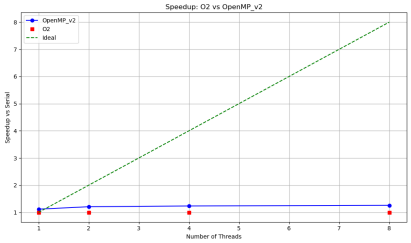
String 500MB



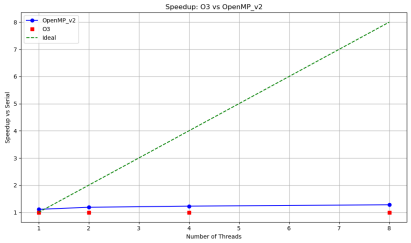
(a) Speedup with no optimization



(b) Speedup with O1 optimization



(c) Speedup with O2 optimization



(d) Speedup with O3 optimization

Figure 10: Speedup for input 500MB

3.6 Conclusion

From the collected measurements, we observe that in the first OpenMP version the execution time worsens compared to the sequential implementation, resulting in a very low speedup, as initially expected. As the number of threads increases, the execution time further increases. This behavior is caused by the structure of the three for loops and by the overhead introduced when spawning and managing a larger number of threads. Moreover, one of the loops contains a critical section: increasing the number of threads increases contention for that section, which further slows down the overall execution.
In the second version, we observe an improvement in performance, achieving a speedup of 1.28. This improvement results from not parallelizing the three for-loops, which reduces the overhead associated with thread management and synchronization. However, the speedup remains moderate because the main loop within the Myers-Mamber algorithm is still executed sequentially, limiting the overall parallel efficiency.

4 CUDA

4.1 Problem description

Provide a parallel version of the Mamber-Myers algorithm with CUDA.

4.2 Solution

As mentioned in Chapter 3.2, the sequential version is used as the basis for developing the parallel version in CUDA. Performance evaluation is carried out by comparing the execution times of the sequential and parallel versions, with the goal of reducing execution time while maintaining the correctness of the suffix array.

4.2.1 Sequential

The sequential version is the same as the one used for OpenMP, described in Chapter 3.2.1.

4.2.2 Parallel

The CUDA version preserves the original *doubling* logic but replaces each sequential step with a parallel equivalent. The algorithm proceeds through the following pipeline in every iteration:

1. **Parallel key generation.** Each suffix i independently computes its comparison key $(\text{rank}[i], \text{rank}[i+h])$. This is implemented with a single `thrust::transform` call, where each GPU thread processes one index.

2. **Parallel sorting.** Instead of manually performing bucket sorting as on the CPU, the keys are sorted using `thrust::sort_by_key`, which executes a highly optimized parallel radix/merge sort on the GPU. This step determines the lexicographic order of the suffixes.

3. **Parallel bucket boundary detection.** After sorting, suffixes that belong to different buckets are detected by comparing adjacent keys. Each thread sets a flag to 1 when the key changes and to 0 otherwise.

4. **Parallel rank reconstruction using prefix sums.** The original CPU version used an array of counters to update ranks inside buckets. This is inherently sequential and not suitable for GPUs. Instead, the CUDA version performs an *inclusive scan* (prefix sum) on the flag array. The result assigns a unique bucket ID (and therefore a new rank) to each suffix in parallel. This replaces both the bucket–head logic and the counter-based refinement of the CPU version.

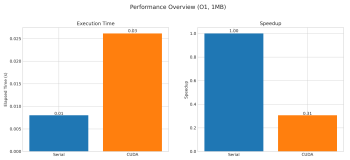
5. **Parallel scatter of new ranks and suffix positions.** Each thread writes the new rank back to its original suffix index and updates the suffix array `pos` accordingly.

4.3 Analysis

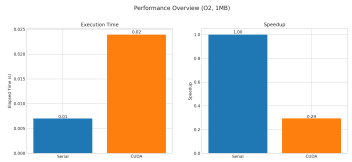
The performance evaluation was conducted by measuring the execution time and the speedup. All tests were performed on strings of increasing size: 1 MB, 50 MB, 100 MB, 200 MB, and 500 MB.

String 1MB

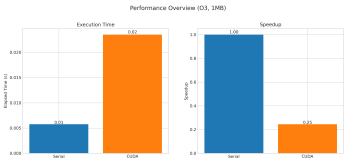
(a) Execution time and speedup with no optimization



(b) Execution time and speedup with O1 optimization



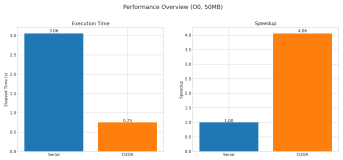
(c) Execution time and speedup with O2 optimization



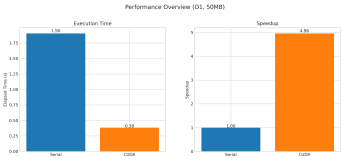
(d) Execution time and speedup with O3 optimization

Figure 11: Execution time and speedup for input 1MB

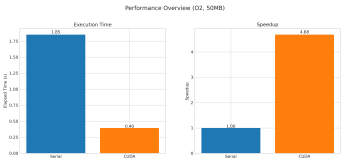
String 50MB



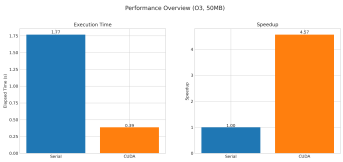
(a) Execution time and speedup with no optimization



(b) Execution time and speedup with O1 optimization



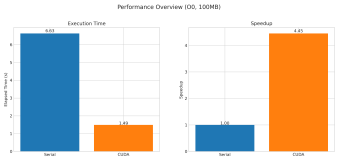
(c) Execution time and speedup with O2 optimization



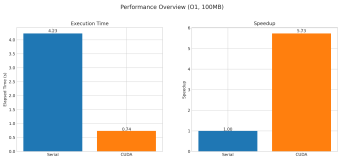
(d) Execution time and speedup with O3 optimization

Figure 12: Execution time and speedup for input 50MB

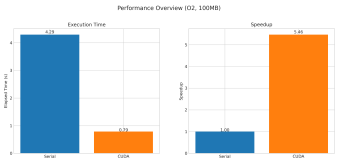
String 100MB



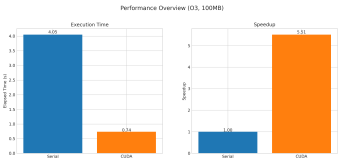
(a) Execution time and speedup with no optimization



(b) Execution time and speedup with O1 optimization



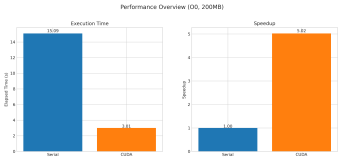
(c) Execution time and speedup with O2 optimization



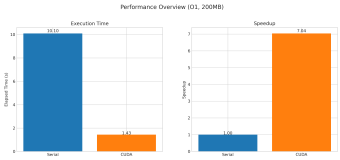
(d) Execution time and speedup with O3 optimization

Figure 13: Execution time and speedup for input 100MB

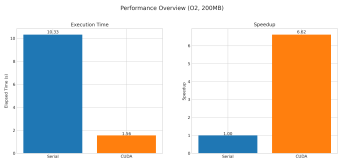
String 200MB



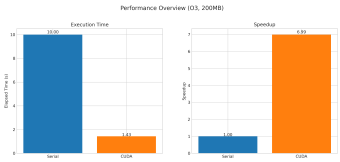
(a) Execution time and speedup with no optimization



(b) Execution time and speedup with O1 optimization



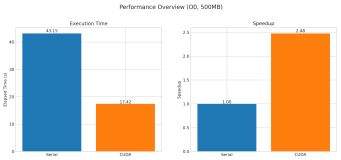
(c) Execution time and speedup with O2 optimization



(d) Execution time and speedup with O3 optimization

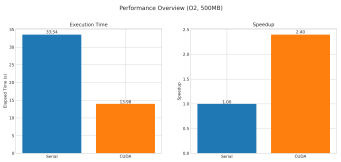
Figure 14: Execution time and speedup for input 200MB

String 500MB

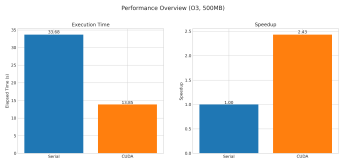


(a) Execution time and speedup with no optimization

(b) Execution time and speedup with O1 optimization



(c) Execution time and speedup with O2 optimization



(d) Execution time and speedup with O3 optimization

Figure 15: Execution time and speedup for input 500MB

4.4 Conclusion

From the measurements, we observed that the CUDA parallel version is significantly faster than the sequential one. This is because the entire doubling logic has been parallelized. In this version, there are no CPU code sections as in the OpenMP version, which results in increased performance for every input. CUDA achieves a speedup of 7.04.

5 How to run

To compile the project and generate executables and plots, use the provided `Makefile`.

Note

Before running any `make` command, ensure that the following tools and libraries are installed:

- `gcc` and `nvcc` compilers
- Python with the `pandas` and `matplotlib` libraries
- `fopenmp` support for compiling OpenMP versions
- The `make` utility

Running `make all` will generate the input strings, all executables, and all plots. **Warning:** this process may take a long time.

Available Make Commands

- `make generate` — generates only the input strings.
- `make exe_serial` — builds the executables of the sequential algorithm with all optimizations.
- `make exe_omp_v1` — builds the executables of the first OpenMP version with all optimizations.
- `make exe_omp_v2` — builds the executables of the second OpenMP version with all optimizations.
- `make exe_cuda` — builds the executables of the CUDA-parallelized version with all optimizations.
- `make run` — runs the sequential version with all optimizations.
- `make run_omp_v1` — runs the first OpenMP version with all optimizations and varying thread counts.
- `make run_omp_v2` — runs the second OpenMP version with all optimizations and varying thread counts.
- `make run_cuda` — runs the CUDA version with all optimizations.
- `make run_graphs_omp_v1` — generates the plots for the OpenMP Version 1 implementation.
- `make run_graphs_omp_v2` — generates the plots for the OpenMP Version 2 implementation.

- `make run_graphs_cuda` — generates the plots for the CUDA implementation.
- `make clean_string` — removes all generated input strings.
- `make clean` — removes strings, measure files, plots, and executables.

6 File Used and Description

- **extractmb.c**: Extracts the number from the input string.
- **generator.c**: Generates input strings and saves them into a file.
- **time.c**: Contains functions to calculate speedup and efficiency.
- **load_string.c**: Reads strings from the input file.
- **main_serial.c**: Main program of the sequential version.
- **main_OpenMP.c**: Main program of the OpenMP parallel version.
- **main_CUDA.cu**: Main program of the CUDA version.
- **suffix_arrays.c**: Sequential implementation of the suffix array.
- **suffix_arrays_OpenMP_v1.c**: First OpenMP version of the suffix array.
- **suffix_arrays_OpenMP_v2.c**: Second OpenMP version of the suffix array.
- **suffix_arrays_CUDA.cu**: CUDA implementation of the suffix array.
- **Graph.py**: Python script to generate graphs for OpenMP.
- **Graph_CUDA.py**: Python script to generate graphs for CUDA.

7 Bibliography and siteography

`https://gist.github.com/sumanth232/e1600b327922b6947f51`