Report Levenshtein distance parallel computing

Veronica Sgamuffa

February 2026

1 Introduction

Levenshtein distance (or edit distance) represents the minimum number of elementary operations (insertion, deletion, or replacement of a character) necessary to transform one string into another. The classic algorithm has a complexity of O(NxM), where N and M are the lengths of the two strings; the computational cost can increase rapidly for large strings (sequential execution on the CPU becomes a bottleneck), making it necessary to adopt parallelization techniques.

$$\operatorname{lev}(a,b)=\begin{cases}|a| & \text{if } |b|=0,\\ |b| & \text{if } |a|=0,\\ \operatorname{lev}\big(\operatorname{tail}(a),\operatorname{tail}(b)\big) & \text{if } \operatorname{head}(a)=\operatorname{head}(b),\\ 1+\min\begin{cases}\operatorname{lev}\big(\operatorname{tail}(a),b\big)\\ \operatorname{lev}\big(a,\operatorname{tail}(b)\big)\\ \operatorname{lev}\big(\operatorname{tail}(a),\operatorname{tail}(b)\big)\end{cases} & \text{otherwise}\end{cases}$$

Figure 1: Levenshtein distance between two strings a and b

The objective of this project is to compare a sequential implementation on CPU and a parallel implementation on GPU using CUDA (exploiting the Wavefront parallelization pattern) of the Levenshtein algorithm, analyzing its performance and the limits of parallelizing an algorithm with strong dependencies between data.

2 Implementation

The first block of code is divided into three main components: a data generation and testing module, a sequential implementation, and a parallel implementation based on the Wavefront pattern.

generateWord is used to create random strings of varying lengths: this function ensures that tests are not influenced by particular character patterns (such as identical or completely different strings) for the purposes of performance analysis.

sequentialDistance implements the Wagner-Fischer algorithm. Instead of allocating a matrix of size (lenA+1)x(lenB+1), only two vectors are maintained: "prev" (previously calculated row) and "curr" (row being calculated). The double "for" loop scans the matrix row by row. The use of std::min allows the minimum cost between deletion, insertion, and replacement to be evaluated, updating the prev vector at the end of each row.

```
int sequentialDistance(std::string A, std::
    string B) {

 int lenA = A.size();
 int lenB = B.size();
 std::vector<int> prev(lenB + 1);
 std::vector<int> curr(lenB + 1);
 for (int j = 0; j <= lenB; j++) prev[j] = j
     ;
 for (int i = 1; i <= lenA; i++) {
   curr[0] = i;
   for (int j = 1; j <= lenB; j++) {
     int cost = (A[i - 1] == B[j - 1]) ? 0 :
          1;
     curr[j] = std::min({curr[j - 1] + 1,
         prev[j] + 1, prev[j - 1] + cost});
   }
   prev = curr;
 }
 return prev[lenB];
}
```

Listing 1: Sequential function

Two different implementations have been developed for the parallel part.

2.1 Implementation A

The complexity of the kernel lies in managing the geometry of the rectangular matrix.

Using equation i + j = k, the kernel calculates a specific antidiagonal. The variables i_min and i_max define the physical limits of the diagonal within the rectangle of the matrix, preventing memory access outside the edges of the strings. Since the diagonals are stored in three linear buffers (pprev, prev, curr), the kernel must translate the two-dimensional coordinates (i, j) into one-dimensional indices. This is done by subtracting the offset max(0, k - len2), which represents the shift of the diagonal when it exceeds the lateral boundaries of the matrix. Each thread calculates its own coordinate i using its tid, compares the corresponding characters, and determines the value of the cell using the results stored on the two previous diagonals.

```
__global__ void levKernel(const char *s1,
    const char *s2, int len1, int len2,
    unsigned int *pprev, unsigned int *prev,
    unsigned int *curr, int k) {

 //i = row, j = column. Diagonal k = i + j
 int i_min = max(1, k-len2);
 int i_max = min(k-1, len1);
 int diagSize = i_max - i_min + 1;

 int tid = blockIdx.x * blockDim.x +
     threadIdx.x;

 if (tid < diagSize) {
  int i = i_min + tid;
  int j = k-i;

  int cost = (s1[i-1] == s2[j-1]) ? 0 : 1;

  //mapping of indexes in the buffers of the
       previous diagonals
  //relative offsets to find (i-1, j), (i, j
      -1), and (i-1, j-1)
  int i_start_prev = max(1, (k-1) - len2);
  int i_start_pprev = max(1, (k-2) - len2);

  unsigned int del = prev[i - 1 -
      i_start_prev + ( (k-1) > len2 ? 1 : 0 )
      ] + 1;
  unsigned int ins = prev[tid + (k > len2 +
      1 ? 1 : 0)] + 1;
  unsigned int sub = pprev[tid + (k > len2 +
       1 ? 1 : ( (k-1) > len2 ? 1 : 0 )) ] +
      cost;

  //indexing simplification
  int idx_prev_up = i - 1 - max(0, (k-1) -
      len2);
  int idx_prev_left = i - max(0, (k-1) -
      len2);
  int idx_pprev_diag = i - 1 - max(0, (k-2)
      - len2);

  curr[i - max(0, k-len2)] = MIN3(prev[
      idx_prev_up] + 1, prev[idx_prev_left] +
       1, pprev[idx_pprev_diag] + cost);
 }
}
```

Listing 2: Kernel CUDA

To optimize the kernel, unsigned int types were chosen, which ensure more efficient memory address calculation and better exposure to NVCC compiler optimizations, avoiding sign handling instructions. At the same time, simplification of relative indexes (such as idx_prev_up, idx_prev_left, and idx_pprev_diag) allows the arithmetic complexity to be moved outside of critical comparison operations, reducing pressure on the GPU registers and minimizing the number of redundant operations for each thread, resulting in an increase in overall throughput during the entire matrix scanning phase.

parallelDistance function allocates buffers (cudaMalloc) and transfers input strings; the size of the buffers (max_dim) is calculated to accommodate the longest possible diagonal, i.e., the one with size lenA+lenB. The main "for" loop scans all lenA+lenB diagonals of the matrix. The kernel is launched only if there are internal cells to calculate: the implementation optimizes the execution flow by limiting the GPU's intervention to only those internal cells that require comparison logic, managing the base cases, i.e., the edges, through direct initializations. Furthermore, the use of pointer rotation for the diagonal buffers at the end of each iteration (tmp = d_pprev; d_pprev = d_prev; ...) allows the entire data set to be kept in memory for the duration of the calculation, thus avoiding costly data transfers between the CPU and GPU.

To explore the concept of diagonal calculation in more depth, while in a square matrix the calculation diagonal increases and decreases symmetrically, in a rectangular matrix (NxM) the wavefront passes through three distinct phases:

1. Growth phase (triangular): the number of cells that can be calculated simultaneously increases with each time step k, until the diagonal reaches the edge of the shortest side of the matrix.

2. Sliding phase (rectangular): when the diagonal reaches a size equal to the shortest string, the number of active threads stabilizes. In this phase, the wavefront no longer expands, but moves sideways along the longest string. Mathematically, this is handled by dynamically updating the row indices: i_min = max(1, k-len2). This ensures that the calculation moves like a sliding window, progressively comparing the characters of the short string with the entire length of the long string.

3. Decreasing phase (triangular): Once the extremity of the diagonal reaches the opposite corner of the matrix, the wavefront begins to decrease until it converges in the last cell (lenA, lenB).

In summary, **parallelDistance** is the function that marks the three phases of the wavefront in time, while **levKernel** is the function that implements the geometric logic and cost calculation within each of them.

The main advantages of the sliding window approach are that the system maintains a constant workload for most of the execution and that, by calculating relative offsets, each thread is able to track its neighbors (above, left, top left) in the linear buffers even if their physical location in memory changes with each sliding step.

```
int parallelDistance(const char *A, const
    char *B, int lenA, int lenB, int
    maxBlockSize) {

  if (lenA == 0) return lenB;
  if (lenB == 0) return lenA;

  char *devA, *devB;
  unsigned int *d_pprev, *d_prev, *d_curr;
  int max_dim = lenA + lenB + 1;

  cudaMalloc(&devA, lenA);
  cudaMalloc(&devB, lenB);
  cudaMalloc(&d_pprev, max_dim * sizeof(
      unsigned int));
  cudaMalloc(&d_prev, max_dim * sizeof(
      unsigned int));
  cudaMalloc(&d_curr, max_dim * sizeof(
      unsigned int));

  cudaMemcpy(devA, A, lenA,
      cudaMemcpyHostToDevice);
  cudaMemcpy(devB, B, lenB,
      cudaMemcpyHostToDevice);

  //edge initialization (diagonal 0 (0,0)
      and Diagonal 1 (1,0 and 0,1))
  unsigned int h_init0 = 0;
  unsigned int h_init1[2] = {1, 1};
  cudaMemcpy(d_pprev, &h_init0, sizeof(
      unsigned int), cudaMemcpyHostToDevice);
  cudaMemcpy(d_prev, h_init1, 2 * sizeof(
      unsigned int), cudaMemcpyHostToDevice);

  for (int k = 2; k <= lenA + lenB; k++) {
    int i_min = max(0, k - lenB);
    int i_max = min(k, lenA);
    int diagSize = i_max - i_min + 1;

    //loading edges (i,0) and (0,j)
    unsigned int val_k = (unsigned int)k;
    if (i_min == 0) cudaMemcpy(&d_curr[0], &
        val_k, sizeof(unsigned int),
        cudaMemcpyHostToDevice);
    if (i_max == k) cudaMemcpy(&d_curr[
        diagSize - 1], &val_k, sizeof(
        unsigned int), cudaMemcpyHostToDevice
        );

    //kernel for internal cells
    int internal_start = max(1, k - lenB);
    int internal_end = min(k - 1, lenA);
    int internal_size = internal_end -
        internal_start + 1;

    if (internal_size > 0) {
      int blockSize = min(internal_size,
          maxBlockSize);
      int gridSize = (internal_size +
          blockSize - 1) / blockSize;
      levKernel<<<gridSize, blockSize>>>(
          devA, devB, lenA, lenB, d_pprev,
          d_prev, d_curr, k);
    }
    cudaDeviceSynchronize();

    unsigned int *tmp = d_pprev; d_pprev =
        d_prev; d_prev = d_curr; d_curr = tmp
        ;
  }

  unsigned int result;
  cudaMemcpy(&result, d_prev, sizeof(
      unsigned int), cudaMemcpyDeviceToHost);

  cudaFree(devA); cudaFree(devB);
  cudaFree(d_pprev); cudaFree(d_prev);
      cudaFree(d_curr);
  return (int)result;
}
```

Listing 3: Parallel function

The run_tests function automates data collection for the report: it runs tests on different string lengths (sizes) and different thread configurations per block (blockSizes). Each test is repeated ”nTests” times and the average execution times are calculated, ensuring that system latency peaks or GPU warm-up phases do not alter the final results. The output returns an array containing the average times and speedups.

2.2 Implementation B

The computation architecture is divided into two kernels to maximize efficiency on the GPU. The first, initLevenshteinMatrix, is responsible for preparing the base cases: instead of initializing the matrix on the CPU and suffering the bottleneck of a huge data transfer to the GPU, this kernel directly sets the first row and first column, i.e., the values dp[i][0] = i and dp[0][j] = j.

```
__global__ void initLevenshteinMatrix(int *
    matrix, int len1, int len2) {

    int idx = blockIdx.x * blockDim.x +
        threadIdx.x;
    int cols = len2 + 1;

    //initialize first row [0][j] = j
    if (idx <= len2) {
        matrix[idx] = idx;
    }

    //Initialize first column [i][0] = i
    if (idx <= len1) {
        matrix[idx * cols] = idx;
    }
}
```

Listing 4: Kernel 1

The second kernel, levKernelCooperative, implements Wavefront logic. Since each cell (i,j) depends on the values above, to the left, and to the upper left, the kernel proceeds diagonally. The distinctive feature here is the use of Cooperative Groups: thanks to grid.sync(), threads can synchronize globally after each diagonal without the kernel having to terminate and restart from the host. This drastically reduces the latency overhead typical of multiple kernel launches, allowing threads to calculate in parallel all cells on a diagonal that do not have mutual dependencies.

```
__global__ void levKernelCooperative(const
    char *s1, const char *s2, int len1, int
    len2, int *matrix) {
    cg::grid_group grid = cg::this_grid();
    int total_threads = gridDim.x * blockDim
        .x;
    int tid = blockIdx.x * blockDim.x +
        threadIdx.x;
    int cols = len2 + 1;

    //cycle on diagonals - wavefront
    for (int k = 2; k <= (len1 + len2); ++k)
         {
        int i_start = max(1, k - len2);
        int i_end = min(len1, k - 1);
        int diag_size = i_end - i_start + 1;

        for (int idx = tid; idx < diag_size;
             idx += total_threads) {
            int i = i_start + idx;
            int j = k - i;
            int cost = (s1[i - 1] == s2[j -
                1]) ? 0 : 1;
            int current_idx = i * cols + j;
            int up    = (i - 1) * cols + j;
            int left  = i * cols + (j - 1);
            int diag  = (i - 1) * cols + (j
                - 1);

            int res = matrix[up] + 1;
            if (matrix[left] + 1 < res) res
                = matrix[left] + 1;
            if (matrix[diag] + cost < res)
                res = matrix[diag] + cost;
            matrix[current_idx] = res;
        }
        grid.sync();
    }
}
```

Listing 5: Kernel 2

The parallelDistance function allocates memory on the GPU for the strings and the cost matrix, then transfers only the strings. Using cudaOccupancyMaxActiveBlocksPerMultiprocessor, the function queries the CUDA driver to determine the ideal number of blocks to launch based on GPU resources, ensuring that the hardware is saturated but not overloaded. Finally, after launching the cooperative kernel, it performs a final synchronization and retrieves only the last integer of the matrix, which represents the calculated Levenshtein distance, then frees all allocated memory to avoid memory leaks.

```
extern "C" int parallelDistance(const char *
    h_s1, const char *h_s2, int len1, int
    len2, int threadsPerBlock) {
  int *d_matrix;
  char *d_s1, *d_s2;
  int size1 = len1 + 1;
  int size2 = len2 + 1;
  size_t matrix_elements = (size_t)size1 *
      size2;
  size_t matrix_bytes = matrix_elements *
      sizeof(int);

  cudaMalloc(&d_matrix, matrix_bytes);
  cudaMalloc(&d_s1, len1);
  cudaMalloc(&d_s2, len2);

  cudaMemcpy(d_s1, h_s1, len1,
      cudaMemcpyHostToDevice);
  cudaMemcpy(d_s2, h_s2, len2,
      cudaMemcpyHostToDevice);

  //initializing matrix on GPU
  int max_dim = (len1 > len2) ? len1 : len2;
  int initBlocks = (max_dim + 255) / 256;
  initLevenshteinMatrix<<<initBlocks,
      256>>>(d_matrix, len1, len2);
  cudaDeviceSynchronize();

  //calculation cooperative
  int numBlocksPerSM;
  cudaOccupancyMaxActiveBlocksPerMultiprocessor
      (&numBlocksPerSM, levKernelCooperative,
       threadsPerBlock, 0);
  int device;
  cudaGetDevice(&device);
  cudaDeviceProp prop;
  cudaGetDeviceProperties(&prop, device);
  int totalBlocks = numBlocksPerSM * prop.
      multiProcessorCount;

  void* args[] = { (void*)&d_s1, (void*)&
      d_s2, (void*)&len1, (void*)&len2, (void
      *)&d_matrix };
  cudaLaunchCooperativeKernel((void*)
      levKernelCooperative, totalBlocks,
      threadsPerBlock, args);

  cudaDeviceSynchronize();

  int result = 0;
  cudaMemcpy(&result, &d_matrix[
      matrix_elements - 1], sizeof(int),
      cudaMemcpyDeviceToHost);

  cudaFree(d_matrix); cudaFree(d_s1);
      cudaFree(d_s2);
  return result;
}
```

Listing 6: Parallel function

3 Performance analysis

The tests were carried out by comparing sequential implementation on CPU with parallel implementation on NVIDIA GPU, varying the length of the strings from 100 to 40.000 characters and testing different block size configurations. Since I have no NVIDIA GPU available on my PC, I wrote and ran the code on Google Colab with a Tesla T4 GPU, the characteristics of which are shown in the following table:

Product name	Tesla T4
Product brand	NVIDIA
CUDA Version	12.4
Driver Version	550.54.15

3.1 Performance analysis A

When analyzing execution times, the sequential algorithm shows quadratic growth. For strings of 100 characters, the calculation time is almost instantaneous (0.4 ms), but it quickly worsens to 74.45 ms (about 74 s) with a word length of 40.000 characters. The parallel version, on the other hand, shows superior scalability. Although the GPU is initially penalized by overhead (30 ms per 100 characters), the growth curve remains much flatter. At a maximum size of 40.000 characters, parallel computation takes only 536 ms, demonstrating very efficient handling of substantial workload.

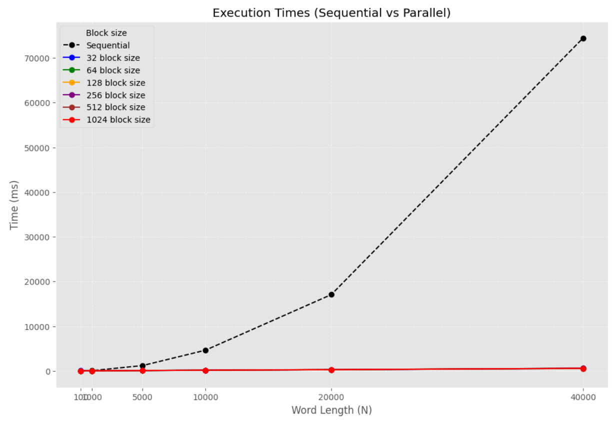

Figure 2: Time - Word length

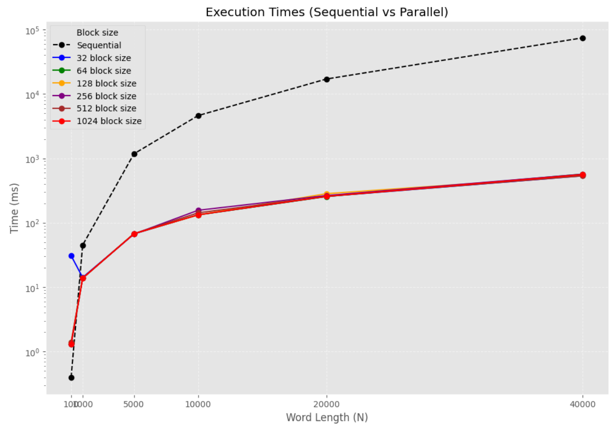

Figure 3: Time - Word length (log)

The speedup highlights the real advantage of parallel architecture. For very small strings (100 characters), the speedup we can achieve is always less than 1 (best speedup 0.31x with block size = 64): the time required to allocate memory on the GPU and transfer data exceeds the actual computation time, making computation with the CPU significantly better. However, with words of 1.000 characters, the GPU already outperforms the CPU with a speedup of about 3x, and the more we increase the size of the problem, the more we notice a progressive improvement (in the case of 5.000, 10.000 and 20.000 characters we obtain a maximum speedup of 17.61x, 34.78x, and 66.66x, respectively). The efficiency of the GPU increases to a maximum speedup of 138.79x with strings of 40.000 characters.

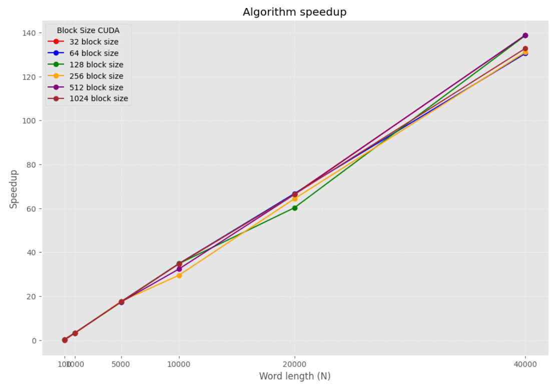

Figure 4: Speedup - Word length

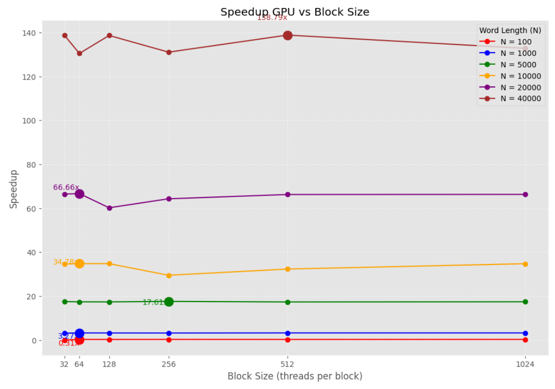

Figure 5: Speedup - Block Size

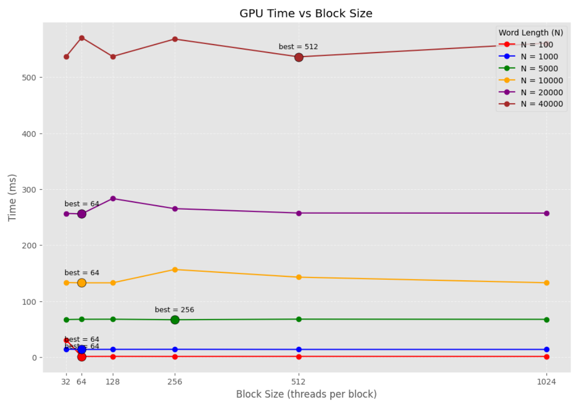

Figure 6: Time - Block size

3.2 Performance analysis B

Analysis of the collected data shows that speedup follows an evolutionary curve linked to the size of the problem. In the initial stages with N=100, parallelism is nearly ineffective due to the dominant overhead generated by memory allocation and kernel launch, operations that exceed the actual calculation time.

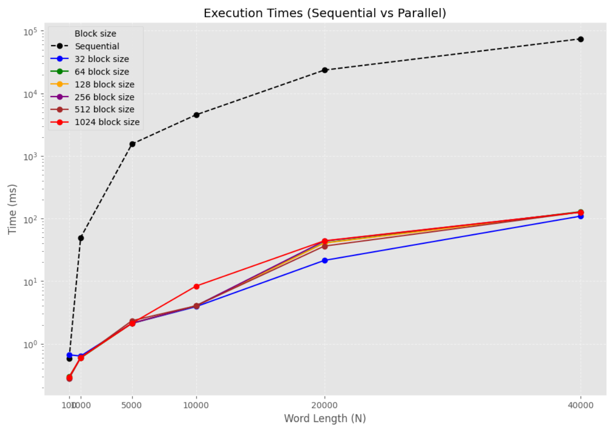

Figure 7: Time - Word Lenght (B)

However, as the workload increases to N=10,000, there is a surge in performance that culminates in an extraordinary speedup peak of approximately 1162x. This value indicates the achievement of the optimal equilibrium point where memory latency is effectively hidden by the high number of arithmetic operations performed in parallel.

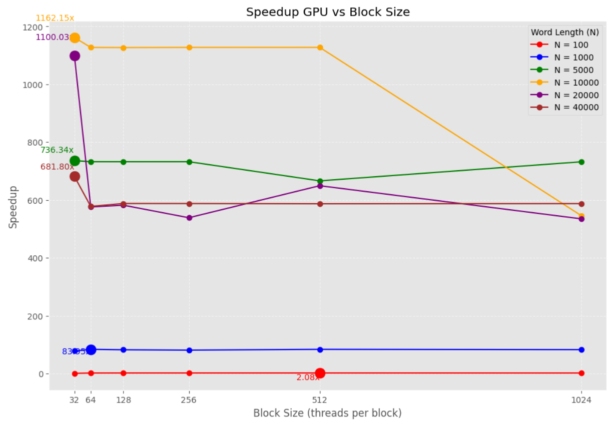

Figure 8: Speedup - Block size (B)

From the analysis of block Size = 32, we note that it was the most efficient and stable, especially in the most demanding configurations. This success is attributable to the perfect coincidence with the size of a CUDA Warp, which minimizes thread divergence and makes global synchronization via Cooperative Groups extremely lighter than larger blocks. In contrast, heavier configurations, such as block size 1024, showed instability and sudden performance drops, probably due to excessive pressure on the registers and more demanding synchronization.

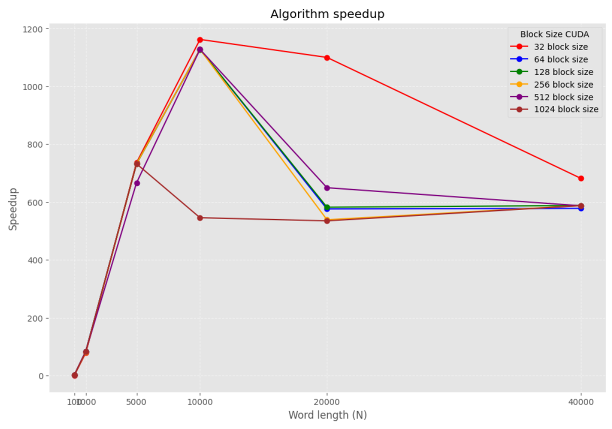

Figure 9: Speedup (B)

Finally, setting N=40.000 results in a natural decline in speedup while maintaining incredibly low execution times compared to sequential execution. This phenomenon marks the transition of the system from a "Compute Bound" to a "Memory Bound" condition, where the matrix probably exceeds the capabilities of the cache, forcing the threads to wait for data from the VRAM and thus limiting the maximum performance gain that can be achieved.

4 Conclusions

4.1 Conclusions A

The project just presented has demonstrated the effectiveness of parallelization with NVIDIA CUDA in solving the Levenshtein distance calculation problem. Through the implementation of the Wavefront pattern for antidiagonals, it was possible to overcome the limitations of the sequential approach, transforming a prohibitively scalable calculation into a process that can be managed in fractions of a second. The stability of the algorithm with respect to block size variation leads us to conclude that the algorithm is not particularly limited by the number of threads per block, but rather by the number of antidiagonals to be processed sequentially. These considerations are made while being aware that this parallelization is not optimal, especially considering the fact that the parallel function calls the kernel many times.

4.2 Conclusions B

The results obtained confirm that the use of Cooperative Groups and global grid synchronization is a good choice for overcoming the limitations of classic multiple kernel launches. Achieving a speedup of more than 1100x highlights the GPU's ability to significantly reduce computation times, making it feasible to analyze large strings (N=40,000) that would be prohibitive in a purely sequential environment. The project also revealed the importance of hardware configuration and resource management, as well as the physical limitations related to memory: it provided empirical evidence of how code optimization must always be based on a deep understanding of the underlying architecture, balancing sophisticated algorithms with real world constraints.