Comparison between sequential and parallel programming on Levenshtein Distance with CUDA

Francesco Todino
E-mail address
francesco.todino@edu.unifi.it

Abstract

The aim of this paper is to compare the performance of sequential and parallel Levenshtein distance algorithm using the CUDA parallel library.

1. Introduction

In this paper, we will analyze the performances obtained by sequential and parallel implementations of the Levenshtein distance algorithm. Our primary focus is to understand the capabilities of the CUDA parallel library to enhance the efficiency of sequential algorithms through GPUs. We will cover different tests, including benchmarking the execution time, measuring speedup and efficiency, and examining scalability across varying input sizes.

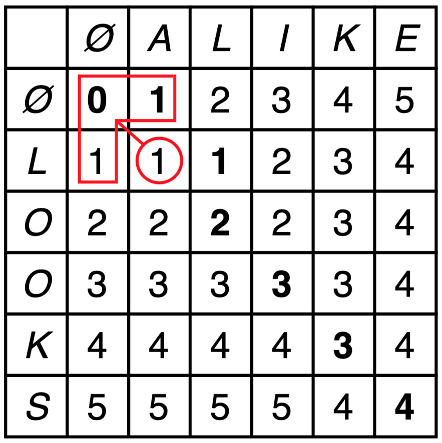
1.1. Levenshtein Distance

To perform these tests we must first take a look at the theory behind Levenshtein Distance. The Levenshtein distance, also known as the edit distance, is a measure of the similarity between two strings in terms of the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into the other. This metric is named after the Soviet mathematician Vladimir Levenshtein, who introduced it in 1965. Mathematically, the Levenshtein distance between two strings *s* and *t* is computed using dynamic programming. The algorithm involves constructing a matrix where each cell *D[i,j]* represents the Levenshtein distance between the prefixes *s[0:i]* and *t[0:j]*. The values in the matrix are filled iteratively based on the minimum cost of the three possible operations; in this case, all operations cost 1. Considering two words *A* and *B* with length *n* and *m* respectively, we start initializing the matrix with:

$$\begin{aligned} D[0,i] = i \quad \forall i = 0...n \\ D[i,0] = i \quad \forall i = 0...m \end{aligned} \tag{1}$$

Then we proceed by defining all the other elements as:

$$D[i,j] = \begin{cases} D[i-1,j-1] & \text{if } A[i] = B[j] \\ 1 + \min \begin{cases} D[i-1,j-1] \\ D[i,j-1] \\ D[i-1,j] \end{cases} & \text{otherwise} \end{cases} \tag{2}$$



	Ø	A	L	I	K	E
Ø	**0**	**1**	2	3	4	5
L	1	1	**1**	2	3	4
O	2	2	**2**	2	3	4
O	3	3	3	**3**	3	4
K	4	4	4	4	**3**	4
S	5	5	5	5	4	**4**

Figure 1. A visual explaination of a matrix obtained with the Levenshtein distance algorithm.

1.2. CUDA

CUDA, or Compute Unified Device Architecture, is a parallel computing platform and programming model developed by NVIDIA. It allows developers to maximize the power of NVIDIA GPUs for general-purpose computing, beyond their traditional role in rendering graphics. CUDA enables programmers to accelerate complex computational tasks by offloading parallelizable portions of their code to the highly parallel architecture of GPUs. CUDA supports popular programming languages like C, C++, and Fortran, allowing developers to seamlessly integrate GPU acceleration into their existing codebases. The ecosystem around

CUDA includes a rich set of libraries, tools, and resources that further simplify the development and optimization of GPU-accelerated applications.

1.3. Setup

All tests were developed in C++ and performed on a Server with an Intel Xeon Silver 4314 with 16 cores and 32 threads and a Nvidia RTX A2000 (12GB) as GPU based on Ampere architecture.

2. Methods

In this section we will see all the code involved into this comparison. Both solutions share only the *generateWord(int)* function which returns a string of the length given in input. The goal was to find the Edit Distance between two strings. To do this, we will follow these steps:

- We create and update a matrix containing all partials distances following the algorithm;
- We return the element in the lower right corner of the matrix, which is the exact distance between the two strings.

Increasing the strings length by only few points can greatly increase the size of the array, so it can become a very memory and processor intensive task.

2.1. Sequential implementation

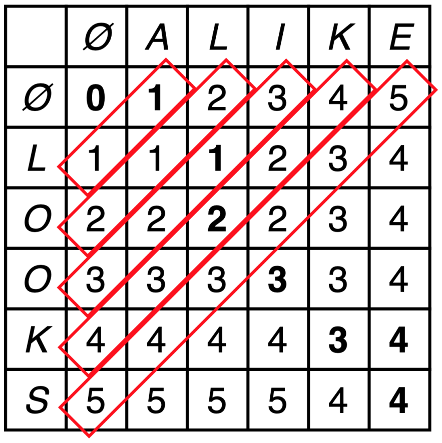
As expected, the sequential version of Edit Distance simply follows the mathematical definition, first defining the first row and column and then iterating the remaining rows sequentially.

```
int sequentialDistance(string A, string B)
{
  unsigned int lenA = A.size();
  unsigned int lenB = B.size();

  unsigned int **D = new unsigned int
      *[lenA + 1];
  for (int i = 0; i < lenA + 1; i++)
    D[i] = new unsigned int[lenB + 1];

  for (int i = 0; i < lenA + 1; i++)
    D[i][0] = i;
  for (int j = 1; j < lenB + 1; j++)
    D[0][j] = j;

  for (int i = 1; i < lenA + 1; i++)
  {
    for (int j = 1; j < lenB + 1; j++)
    {
      if (A[i - 1] == B[j - 1])
      {
        D[i][j] = D[i - 1][j - 1];
      }
      else
      {
        D[i][j] = 1 + min(min(D[i -
            1][j], D[i][j - 1]), D[i -
            1][j - 1]);
      }
    }
  }

  return D[lenA][lenB];
}
```

Listing 1. Sequential implementation

2.2. Sequential implementation

As we can see in Listing 2, to express the full potential of CUDA, we must change the development paradigm while trying to optimize an algorithm.
CUDA works by dividing the code into two parts: one dedicated to the CPU and one dedicated to the GPU.
The one dedicated to the GPU is defined as a kernel function and specified with *__global__* before the function definition.
In CUDA kernel functions we can only use functions and classes from their libraries, so we have to define all the necessary functions within the editDistance function. For example, we even need to use char* instead of std::string.
This algorithm is a tricky one to improve: every iteration uses the elements found in three neighboring but non-consecutive cells according to the matemathical definition. This means that following the same paradigm cannot improve performance on CUDA, considering also that the actual computation is so simple that it does not even require a GPU.



	Ø	A	L	I	K	E
Ø	**0**	**1**	2	3	4	5
L	1	1	**1**	2	3	4
O	2	2	**2**	2	3	4
O	3	3	3	**3**	3	4
K	4	4	4	4	**3**	**4**
S	5	5	5	5	4	**4**

Figure 2. How the optimization is performed.

In order to perform a good optimization, we need to consider which elements can be calculated simultaneously:

since we use the element at the top, left, and top left of the current one, we can consider calculating the elements by iterating the antidiagonals of the matrix instead of the rows of the matrix, since we always calculate the one at the top left in the previous iteration.
In this way, we can perform the best possible optimization, improving as word length increases.

```
int parallelDistance(const char *A, const
    char *B, int lenA, int lenB)
{
  int distance;
  char *devA;
  char *devB;
  unsigned int *devCurrDiag;
  unsigned int *devPrevDiag;
  unsigned int *devPPrevDiag;

  unsigned int *currDiag = new unsigned
      int[lenA + 1];
  unsigned int *prevDiag = new unsigned
      int[lenA + 1];
  unsigned int *pprevDiag = new unsigned
      int[lenA + 1];
  pprevDiag[0] = 0;
  prevDiag[0] = 1;
  prevDiag[1] = 1;

  cudaMalloc((void **)&devA, (lenA + 1) *
      sizeof(char));
  cudaMalloc((void **)&devB, (lenB + 1) *
      sizeof(char));
  cudaMalloc((void **)&devCurrDiag, (lenA
      + 1) * sizeof(unsigned int));
  cudaMalloc((void **)&devPrevDiag, (lenA
      + 1) * sizeof(unsigned int));
  cudaMalloc((void **)&devPPrevDiag, (lenA
      + 1) * sizeof(unsigned int));

  cudaMemcpy((void *)devA, (void *)A,
      (lenA + 1) * sizeof(char),
      cudaMemcpyHostToDevice);
  cudaMemcpy((void *)devB, (void *)B,
      (lenB + 1) * sizeof(char),
      cudaMemcpyHostToDevice);
  cudaMemcpy((void *)devPPrevDiag, (void
      *)pprevDiag, (lenA + 1) *
      sizeof(unsigned int),
      cudaMemcpyHostToDevice);
  cudaMemcpy((void *)devPrevDiag, (void
      *)prevDiag, (lenA + 1) *
      sizeof(unsigned int),
      cudaMemcpyHostToDevice);

  for (int i = 3; i < 2 * (lenA + 1); i++)
  {
    int blockSize = min(i, 1024);
    int gridSize = ceil(float(i) /
        blockSize);

    editDistKernel<<<gridSize,
        blockSize>>>(devA, devB, lenA,
        lenB, devPPrevDiag, devPrevDiag,
        devCurrDiag, i);

    unsigned int *tmp = devPPrevDiag;
    devPPrevDiag = devPrevDiag;
    devPrevDiag = devCurrDiag;
    devCurrDiag = tmp;
  }
  cudaMemcpy((void *)&distance, (void
      *)&devPrevDiag[0], 1 *
      sizeof(unsigned int),
      cudaMemcpyDeviceToHost);

  cudaFree(devA);
  cudaFree(devB);
  cudaFree(devPPrevDiag);
  cudaFree(devPrevDiag);
  cudaFree(devCurrDiag);

  cout << "Distance: " << distance << endl;
  return distance;
}
```

Listing 2. Parallel implementation

In the Listing 3 we can see the implementation of the kernel function and note both the use of the MIN macro and all the references to relevant variables as input.

```
__global__ void editDistKernel(char *devA,
    char *devB, int lenA, int lenB,
    unsigned int *devPPrevDiag, unsigned
    int *devPrevDiag, unsigned int
    *devCurrDiag, int diagIdx)
{
  int tid = blockIdx.x * blockDim.x +
      threadIdx.x;
  int diagSize = (diagIdx <= lenA + 1) ?
      diagIdx : ((2 * (lenA+1)) - diagIdx);

  if (tid < diagSize) {
    if (diagIdx <= lenA + 1) {
      if (tid == 0)
        devCurrDiag[tid] =
            devPrevDiag[tid] + 1;
      if (tid == diagSize - 1)
        devCurrDiag[tid] =
            devPrevDiag[tid - 1] + 1;

      if (tid > 0 && tid < diagSize - 1) {
        if (devA[tid - 1] !=
            devB[diagSize - tid - 2])
          devCurrDiag[tid] = 1 +
              MIN(devPrevDiag[tid - 1],
                MIN(devPPrevDiag[tid - 1],
                devPrevDiag[tid]));
          else
            devCurrDiag[tid] =
                devPPrevDiag[tid - 1];
        }
      } else {
        int pprevIdx = (lenA - diagSize ==
            0) ? tid : tid + 1;
        if (devA[tid + lenA - diagSize] !=
            devB[lenA - tid - 1]) {
          devCurrDiag[tid] = 1 +
              MIN(devPrevDiag[tid],
              MIN(devPPrevDiag[pprevIdx],
              devPrevDiag[tid + 1]));
        }
        else {
          devCurrDiag[tid] =
              devPPrevDiag[pprevIdx];
        }
      }
    }
}
```

Listing 3. CUDA kernel implementation

3. Tests

The performance evaluation tests are aligned with the objectives of the study:

- The first test involves assessing performance by incrementally increasing the length of the words used to evaluate the distance. For this test, a blockSize of 1024 is chosen;

- The second test focuses on comparing performance while varying the blockSize and simultaneously increasing the length of the words.

To avoid biased results, based on the initial CUDA warmup, we tested everything more times and get the average results.

3.1. Increasing words length

The results of the first test, where the words length increases up to 50000, are shown in Figure 3.

As expected, time increases exponentially from 0 ms up to 42577 ms in the sequential version while reaching only 296 ms in the parallel version with 50000 characters (this big difference is why parallel times always seem to be 0).

Then we can see related speedups shown in Figure 4.

Going by the times, the speedups achieved are incredible, having reached speedup up to 143.841x while increasing at every iteration.

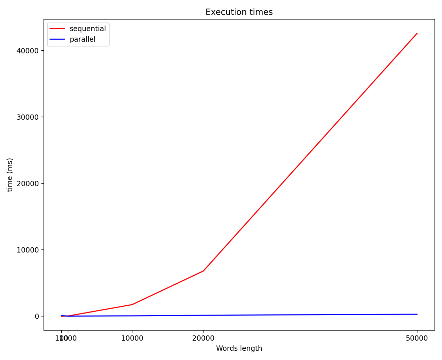


Figure 3. Times finding words distance, increasing words length.

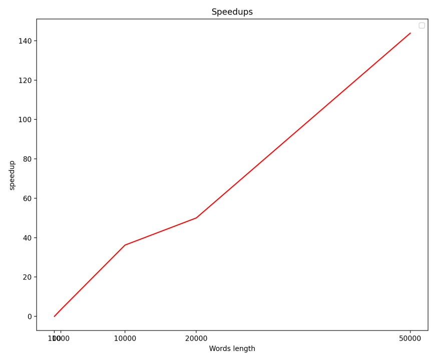


Figure 4. Speedups finding words distance, increasing words length.

3.2. Increasing words length and blockSize

The results of the second test, where the number of blockSize increases and the words length is increased up to 50000, are shown in Figure 5. The performance of all versions aligned somewhat with expectations, excelling particularly with intermediate-sized blocks (such as 256 and 512)

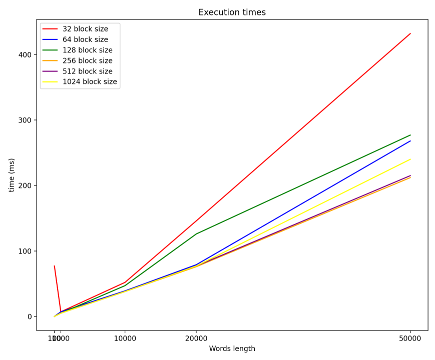


Figure 5. Times finding words distance, increasing words length.

while showing reduced efficiency with smaller blocks. This underscores the importance of striking an optimal balance in the development of parallel solutions.

Then we can see related speedups shown in Figure 6.

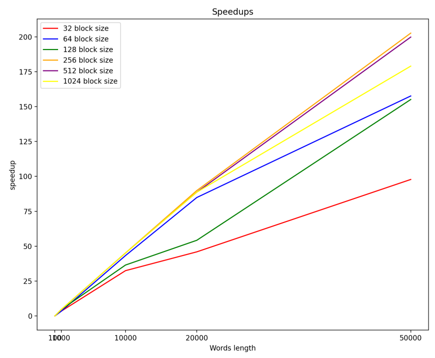


Figure 6. Speedups finding words distance, increasing words length and blockSize.

The best performance obtained is with 256 as blockSize, which reaches an incredible speedup of 202.693x.

4. Conclusions

In conclusion, the comparison between sequential and parallel programming in the context of the Levenshtein Distance using CUDA confirms that the best approach is definitively the parallel one, having obtained outstanding performance in all tests carried out. This experiment teaches us that the effort to think of other ways to solve a problem can lead to very important improvements.