Parallel Needleman-Wunsch on CUDA to measure word similarity based on phonetic transcription

Dominic Plein

February 19, 2025

Part of the GP-GPU Computing course at UGA by Christophe Picard

Abstract

We present a method to calculate the similarity between words based on their phonetic transcription using the Needleman-Wunsch algorithm. We implement this algorithm in Rust and parallelize it on both CPU and GPU to handle large datasets efficiently. The GPU implementation leverages CUDA and the cudarc Rust library to achieve significant performance improvements. We validate our approach by constructing a fully-connected graph where nodes represent words and edge have weights according to the similarity between the words. This graph is then analyzed using clustering algorithms to identify groups of phonetically similar words. Our results demonstrate the feasibility and effectiveness of the proposed method in analyzing the phonetic structure of languages. It might be easily expanded to other languages.

Contents

1 Introduction

The International Phonetic Alphabet (IPA) uses special symbols[1] to represent the sound of a spoken language. This is useful for language learners since the pronunciation of a word can be significantly different from its written form. For example, the French word *renseignement* (information) is pronounced /ʁɑ̃.sɛɲ.mɑ̃/. Based on this alphabet, one might wonder if we can construct a metric that quantifies the **distance between two words based on their phonetic transcription**. This would allow to construct a graph where nodes are words and undirected edges are weighted by the distance between the words. This graph could then be used to find neighbors of a word based on their phonetic similarity. This opens up the possibility to apply clustering algorithms and other methods stemming from graph theory in order to analyze the phonetic structure of a language.

Calculating the distance between each pair of words corresponds to a fully-connected graph. Our dataset consist of around 600,000 French words and their IPA transcription, alongside their frequency in the French language. Excluding self-loops, we find a vast number of edges (14):

$$\#\text{nodes} = 600,000 \tag{1}$$

$$\#\text{edges} = \frac{600,000 \cdot 599,999}{2} \approx 1.8 \times 10^{11} \tag{2}$$

This high number and the independent nature of the distance calculation for each pair of words makes the problem well-suited for parallelization. In section 2, we present the Needleman-Wunsch algorithm used to calculate the distance between two words. In section 4, we discuss how to parallelize this algorithm on a CPU using the *Rayon* library in Rust and on a consumer Nvidia GPU using the CUDA framework with the *cudarc* Rust library. Finally, we provide visualizations of the obtained graphs in 5 and conclude in section 6.

[1] See for example the French list here.

2 Needleman-Wunsch

The Needleman-Wunsch algorithm (developed by Saul B. Needleman and Christian D. Wunsch in 1970) calculates the global alignment of two strings and was originally used in bio-informatics to compare DNA sequences. For our purposes, the alphabet will instead consist of the phonetic IPA symbols. Out of all possible alignments of two words (including gaps), the Needleman-Wunsch algorithm finds the one with the smallest distance, i. e. the alignment with the highest "score". The algorithm is based on dynamic programming and has a time complexity of $\mathcal{O}(\text{len}(A)\cdot\text{len}(B))$, where A and B are the two words to be compared.

Algorithm 2.1 features the pseudo-code of the score computation[2]. In Figure 1, we see the resulting score matrix that the algorithm constructed for the French words *puissance* and *nuance*. Follow the indicated path (red tiles) from the bottom right to the top left to find the (reversed) optimal alignment (see Table 1);

puissance	p	ɥ	i	s	ɑ̃	s
nuance	n	ɥ	–	–	ɑ̃	s

Table 1: The optimal alignment yields a score of −2. See the path in Figure 1.

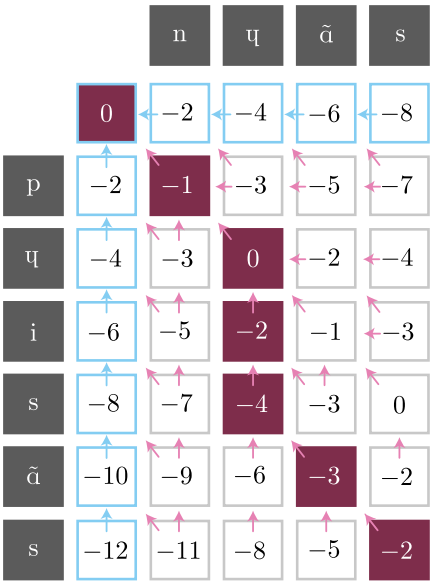

Figure 1: Needleman-Wunsch score matrix for the words $A :=$ *puissance* /pɥisɑ̃s/ (power, strength) and $B :=$ *nuance* /nɥɑ̃s/ (nuance, shade). The arrows indicate which steps locally maximize the score. The red tiles trace the path of the optimal alignment. Match Score: 1, Mismatch Score: −1, Gap Penalty: $p = -2$.

We first discuss the meaning of the different steps (arrows) in the score matrix (Figure 1) to then explain how to construct this matrix.

- In a **diagonal step**, both symbols that indicate the current position in the two words change. Such a step corresponds to either a match or a mismatch between the two symbols. In the example, the /s/ symbols in the bottom-right corner match, which is why the step beforehand is a *diagonal* step from the field −3 to −2. The score increases by 1 since we defined the match score to be +1 (and a mismatch score as −1).
- In a **vertical** or **horizontal step**, only one of the two symbols changes. We interpret this as a gap in the alignment, i. e. one symbol aligns to a gap in the other word. In the example, this is the case two times when we move from the red field 0 down to −2 and then down to −4. The score decreases by 2 each time, as we defined the gap penalty as $p := -2$ in this example. The gap is indicated by "–" in the alignment (see Table 1). As we are still in the column of /ɥ/ of the word /nɥɑ̃s/, we insert two "–" symbols after the /ɥ/ in Table 1. This step is sometimes also referred to as **deletion** or **insertion**.

To find the score matrix for given input words A and B, we follow Algorithm 2.1. First, the score matrix of dimension $(\text{len}(A)+1)\times(\text{len}(B)+1)$ is initialized[3]. Then, in lines 3 to 6, the blue-bordered tiles of Figure 1 are filled with the gap penalty p times the index. This is necessary since the only possible step for these tiles is either a vertical or horizontal step (blue arrows), thus leading to a gap in the alignment as discussed beforehand that we punish with the gap penalty p.

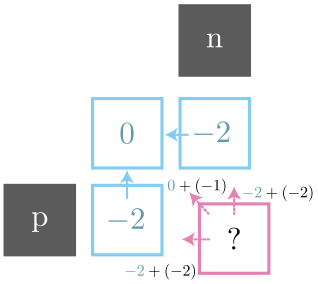

Figure 2: Calculations for one element of the Needleman-Wunsch score matrix.

In the nested loops (lines 7 and 8), we then iterate over the remaining fields of the score matrix (index now starts at 1, not 0) which corresponds to traversing the matrix row-wise. To each field, we assign the maximum of three values:

[2] The respective Wikipedia page also provides a good introduction. Furthermore, the score matrix is interactively explained in the Global Alignment App.

[3] This does not necessarily involve setting all fields to 0 as will become clear.

Algorithm 2.1: Needleman-Wunsch

Input: $A = \{A_0, \ldots, A_{\texttt{len}(A)-1}\}$, $B = \{B_0, \ldots, B_{\texttt{len}(B)-1}\}$, similarity: similarityScoreFunc, p: GapPenalty
Output: score

```
1  Function calculateScore():
2      Init scoreMatrix with dimensions
         (len(A) + 1) × (len(B) + 1)
3      for i ∈ {0, ..., len(A)} do
4          scoreMatrix[i][0] ← p · i
5      for j ∈ {0, ..., len(B)} do
6          scoreMatrix[0][j] ← p · j
7      for i ∈ {1, ..., len(A)} do
8          for j ∈ {1, ..., len(B)} do
9              cost ← similarity(A_{i-1}, B_{j-1})
10             matchScore ← scoreMatrix[i − 1][j − 1] + cost
11             deleteScore ← scoreMatrix[i − 1][j] + p
12             insertScore ← scoreMatrix[i][j − 1] + p
13             scoreMatrix[i][j] ←
                 max(matchScore, deleteScore, insertScore)
14     return scoreMatrix[len(A)][len(B)]
```

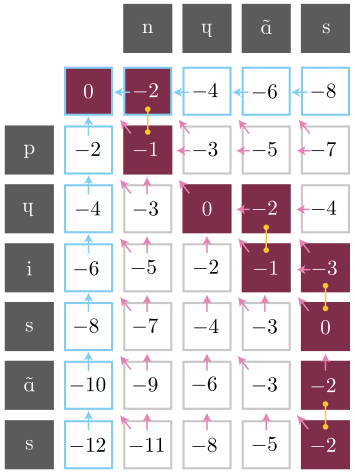

Figure 3: Needleman-Wunsch score matrix and the path (in red) for a non-optimal alignment. Orange strokes indicate non-optimal steps. Parameters as in Figure 1.

- The **match score** is calculated by checking the step to the upper left diagonal (line 10). In the example of Figure 2, this would result in a value $0 + (-1) = -1$, where 0 is the value in the upper left diagonal field and -1 is the cost of the mismatch between /p/ and /n/. In case of a match, the new value would be $0 + 1 = 1$. In the algorithm, we also consider the case where costs for a match and a mismatch depend on the symbols themselves, which is why we introduce the function `similarity` that returns the cost of aligning two symbols. This is especially useful when comparing phonetic symbols, as the similarity between two symbols can be defined in a more sophisticated way than just 1 or -1 (e. g. replacing a vowel with a consonant might be more costly than replacing a vowel with another vowel).

- The **delete score** refers to the step from the field above (line 11). In the example, we find $(-2) + (-2) = -4$ as new value (-2 is the value in the field above and $p = -2$ is the gap penalty). This steps signifies that a symbol in word A aligns to a gap in word B (here: /i/ and /s/ of *puissance* align to gaps in *nuance*).

- The **insert score** refers to the step from the left (line 12). In the example, we find $(-2) + (-2) = -4$ as new value (-2 is the value in the field to the left and $p = -2$ is the gap penalty). This steps signifies that a symbol in word B aligns to a gap in word A (this does not occur in the example).

  The new value of the current field is assigned the maximum of these values (line 13), such that we locally maximize the score: $\max(-1, -4, -4) = -1$. In Figure 1, we additionally kept track of the steps that led to the optimal alignment by means of the rose arrows (here only the diagonal step that yields the new maximal score of -1). For our purposes, we don't want to reconstruct the exact alignment that led to the optimal score, but only the score itself. Thus, we can omit the backtracking step and don't need to store the rose arrows.

By construction, **the bottom-right field of the score matrix contains the score of the optimal alignment**. This is ensured by the Principle of Optimality (Bellman), which states that an optimal solution to a problem can be constructed from optimal solutions to its subproblems. In the context of the Needleman-Wunsch algorithm, this means that the optimal alignment score for two sequences (words) can be derived by considering the optimal alignment scores of progressively smaller subsequences. Each cell in the score matrix represents the optimal score for the corresponding prefixes of the two words up to that point, since we take the maximum of the three possible steps (match, delete, insert) at each cell. This ensures that the final cell (in the bottom right) contains the optimal score for the entire sequences.

Table 2 shows an example of a non-optimal alignment of the two words, yielding a score of -15 (compared to -2 for the optimal path). Figure 3 depicts the corresponding score matrix. Note how the indicated path includes 4 non-optimal choices (orange strokes).

puissance	–	p	ɥ	–	i	–	s	ɑ̃	s
nuance	n	–	ɥ	ɑ̃	–	s	–	–	–

Table 2: This non-optimal alignment yields a score of -15. See the path in Figure 3.

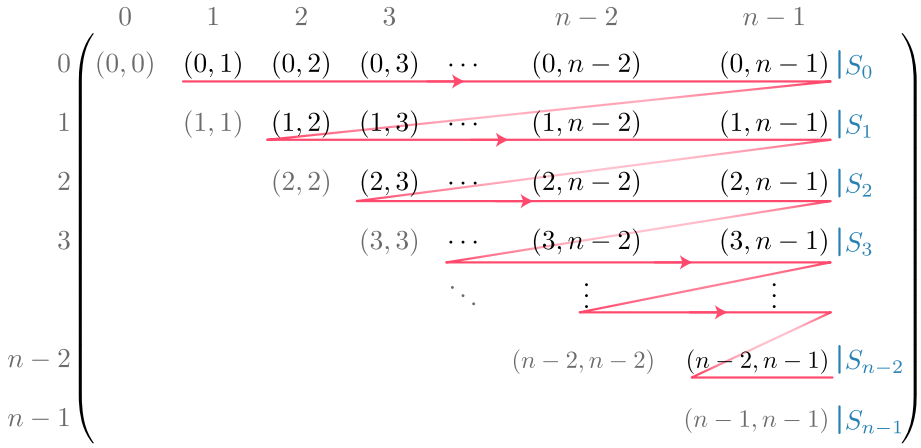

Figure 4: Row-major traversal of the adjacency matrix. n is the total number of words (i. e. nodes).

3 Data Preparation

We use the **french-words** dataset (frodonh, 2020), which contains 691,969 French words. It is compiled from several sources including (among others) the Debian package wfrench (used for spell checking), Lexique 3.83 (Boris New & Christophe Pallier), the DELA dictionary from the University of Marne-la-Vallée as well as the French-Dictionary (Hussem Ben Belgacem). The words also include Part-of-Speech (POS) tagging information, e. g. whether it is a noun, verb, adjective, preposition etc. It also comprises the usage frequency according to Lexique.org and Google Ngrams. We use the average of both sources (or just one if the other is missing).

Since the french-words dataset does not include the phonetic transcriptions, we consult the **wiki-pronunciation-dict** (Daniel Wolf, 2021) extracted from the French Wiktionnaire. We merge both datasets to obtain 611,786 words with their IPA transcription. If multiple phonetic transcription are available, we only store the first one. For easy access, we store the data in a dataclass and serialize it to a pickle file of around 60 MB.

Additionally, we extract all used phonetic symbols and assign integer IDs to them. This enables us to store the transcription as a list of integers. For our examples, we obtain Table 3. The Needleman-Wunsch algorithm will then work on these integer lists. Note that we consider /dʒ/ as one symbol, even though it is a combination of /d/ and /ʒ/. The same applies to /tʃ/. This is to account for the different pronunciation of the combined symbols compared to the individual ones.

Word	**IPA**	**Encoding**
puissance	/pɥisɑ̃s/	$[0, 18, 16, 11, 26, 11]$
nuance	/nɥɑ̃s/	$[29, 18, 26, 11]$

Table 3: Example of two words with their phonetic transcription and encoding.

4 Parallelized algorithms

In the following implementations, we always use a similarity matrix with 1 on its diagonal (symbols match) and -1 elsewhere (symbols do not match). The gap penalty is set to $p = -1$ (instead of -2 beforehand in the examples in section 2).

4.1 On the CPU

First, we implement the Needleman-Wunsch algorithm in Python using `numpy`. This implementation serves well to deepen the understanding of the algorithm and to eliminate index errors. However, it is very slow, making it unsuitable for the whole dataset. We then translate the code to an unparallelized Rust version and confirm correct output by direct comparison with the Python implementation. To make use of all CPU cores, we **parallelize the Rust implementation** employing the `rayon` library to parallelize the outer for-loop (line 7): for every word A, we consider all other words B and calculate the similarity between A and B. Since words lengths differ, the core calculation takes different times for different word pairs. Therefore, every thread appends its result to a vector that is wrapped in a `Mutex` (avoid data races by mutual exclusion) and an `Arc` (thread-safe reference pointer to deallocate data at the end).

Since comparing A to B yields the same score as the comparison of B to A, we deal with *undirected* edges and thus the adjacency matrix is symmetric. Furthermore, we are not interested in self-loops, but only in the similarity between *different* words. For these two reasons, **we only consider the upper triangular part of the adjacency matrix** in Figure 4. To store the resulting edge weights in a binary file, we traverse the matrix in row-major order (red path): $(0,1)$, $(0,2)$, …, $(0,n-1)$, $(1,2)$, …, $(1,n-1)$, $(2,3)$, …, $(n-2,n-1)$, where n is the total number of words. Note that in the parallelized Rust implementation, we have to also

return the indices of the words since the order in which the threads terminate is not deterministic (from point of view of the Rust code). After all threads are finished, the results are sorted according to row-major ordering to match the traversal order. When saving the edge weights in a file, we drop the indices and only store the similarity scores as 8-bit signed integers (range $[-128, 127]$). This is sufficient since word lengths are typically small and match/mismatch score as well as gap penalty p are likewise chosen to be small.

4.2 On the GPU

We implement the algorithm in the CUDA framework. We consult the `cudarc` Rust library, which provides Rust wrappers around the CUDA driver API as well as the NVRTC API (among others). The latter makes available a method to compile our C++ kernel to Parallel Thread Execution (PTX) code during runtime and to launch it afterwards.

Subtasks. Foster's methodology can help in designing parallel algorithms. The first step is to partition the problem at hand into small tasks. At the level of the adjacency matrix (Figure 4), such a task would be to compute the similarity score between two words A and B at the respective row and column. On a finer granularity, we can also refer to Figure 2 and consider the calculation of one element of the score matrix. However, we realize that calculations of elements in the score matrix are highly dependent on each other since the score of a cell depends on the scores of its up, left and up-left (diagonal) neighbors (example of a *stencil computation*). Additionally, words length differ, so the size of the score matrix varies. Due to the data dependencies and the varying problem size, we decide to focus on parallelizing the for-loops in the Needleman-Wunsch algorithm (lines 7 and 8 in Algorithm 2.1) making this a *pleasingly parallel* problem, namely that of parallelizing score calculation for entries in the adjacency matrix.

Grid. The maximum number of blocks in a grid (grid size) is limited to $2^{16} - 1 = 65535$ in y and z direction. Starting with CUDA compute capability 3.0, the limit in the x direction is raised to $2^{31} - 1 = 2,147,483,647$ blocks. Furthermore, we only consider the upper triangular part of the adjacency matrix, while a grid is always rectangular. These constraints motivate the choice of using a one-dimensional grid and spawning blocks only in the x direction. Likewise, we only employ one-dimensional blocks, i. e. a block is a one-dimensional array of threads. The exact number of threads we use inside a block (block dimension) is discussed later. With this configuration, we calculate (in every thread) an **index** ranging from 0 to a number great than that of entries in the upper triangular part of the adjacency matrix:

$$\text{idx} = \text{blockDim.x} \cdot \text{blockIdx.x} \cdot \text{threadIdx.x} \tag{3}$$

Row & Column. The index `idx` needs to be mapped to the corresponding row and column in the adjacency matrix, such that a thread knows which words to compare. We take advantage of the row-major traversal path. Let S_r be the number of elements traversed up to the end of row r. In Figure 4, this variable is drawn in blue. Figure 5 provides an example to ease keeping track during following calculations.

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} \cdot & 0 & 1 & 2 \\ & \cdot & 3 & 4 \\ & & \cdot & 5 \\ & & & \cdot \end{pmatrix} \end{array} \begin{array}{l} \\ S_0 = 3 \\ S_1 = 5 \\ S_2 = 6 \\ \\ \end{array}$$

Figure 5: Example of idx (black) for $n = 4$. S_r is the number of elements traversed up to the end of row r (blue).

$$S_0 = n - 1, \quad S_1 = S_0 + (n - 2), \quad \dots \tag{4}$$

$$S_r = \sum_{k=0}^{r} (n - k - 1) \tag{5}$$

$$= (n-1) \cdot (r+1) - \sum_{k=0}^{r} k \tag{6}$$

$$= (n-1) \cdot (r+1) - \frac{r \cdot (r+1)}{2} \tag{7}$$

$$= -\frac{1}{2} r^2 + r\left(n - \frac{3}{2}\right) + (n-1) \tag{8}$$

For a given row r, we have $\text{idx} \in [S_{r-1}, S_r)$. We therefore require $S_{r-1} \stackrel{!}{=} \text{idx}$ and solve for r. After some algebraic manipulations, we obtain:

$$0 = -\frac{1}{2} r^2 + r \underbrace{\left(n - \frac{1}{2}\right)}_{z} - \text{idx} \tag{9}$$

$$\Leftrightarrow 0 = r^2 + r(-2z) + 2 \cdot \text{idx} \tag{10}$$

$$\Rightarrow r(\text{idx}) = z - \sqrt{z^2 - 2 \cdot \text{idx}} \tag{11}$$

We only use the negative branch of the square root since we want the row to increase with increasing index. Furthermore, we require the row index to be an integer, so we round down $\lfloor r(\text{idx}) \rfloor$ such that the row stays the same for a range of increasing indices. We can now also calculate column c:

$$c(\text{idx}) := r + 1 + (\text{idx} - S_{r-1}) \tag{12}$$

$$= \frac{1}{2} r^2 + r\left(\frac{3}{2} - n\right) + (\text{idx} + 1) \tag{13}$$

Finally, note that S_{n-1} conveniently gives us the number of elements in the upper triangular part of the adjacency matrix, i. e. the number of edges in our graph. With (8), we find:

$$\text{num edges} = S_{n-1} = \frac{1}{2} n^2 - \frac{1}{2} n = \frac{n(n-1)}{2} \tag{14}$$

With (11) and (13), we determine the respective row and column in the adjacency matrix for a

given index (3). During this row and column calculation, one should use 64-bit floating point numbers to mitigate precision errors that yield to wrong indices. Furthermore, we deal with graphs of more than 2^{32} edges and thus need to store the `idx` as an 64-bit unsigned integer. The `blockIdx.x` has to be statically cast to `u64` (unsigned long long int), otherwise it is assumed to be `u32` leading to incorrect indices and hard-to-debug errors for big graphs.

Memory. In the first version of our CUDA kernel, for every thread we allocate an array of arrays on the heap inside global GPU memory for the score matrix. We deallocate it after the alignment score is computed. This is very costly and immensely slows down the kernel execution. Instead, we want to use *shared memory* for all threads in a block and let them share one big score matrix as to minimize the number of allocations needed. However, the score matrix dimensions differ in size depending on the word lengths of A and B. Our solution is to look up the maximum word length q before the kernel is launched and allocate enough shared memory, i. e. for every thread, we assume the worst-case scenario of a $(q+1) \times (q+1)$ matrix and allocate $(q+1)^2 \cdot \text{blockDim.x}$ chars (`i8`) as linear shared memory. Then, inside each thread, we retrieve a pointer to the start of the memory reserved for this thread by adding $\text{threadIdx.x} \cdot (q+1)^2$ to the base pointer. To finally access element (i, j) in the score matrix, we calculate $i \cdot \big(\text{len}(B)+1\big) + j$. This may not use all allocated memory, but it is a trade-off between memory usage and performance; with our approach we avoid bank conflicts.

Block dimension. This dimension specifies the number of threads inside a block and is limited to 1024. We want to maximize shared memory available per block in order to minimize the number of memory allocations needed. Before the kernel is launched, for each $u \in [1, 1024]$, the resulting shared memory size for one block ($u \cdot (q+1)^2$ bytes) is calculated. The maximum u is chosen such that the maximum available shared memory per block (retrieved from CUDA device properties) is not exceeded.

5 Evaluation

We deploy our GPU code on a consumer Nvidia GeForce GTX 1060[4] with 6GB GDDR5. Every code change related to the GPU code is verified by comparing the resulting binary edge weights file with the one generated by our parallelized Rust implementation on the CPU. This baseline helps to quickly identify errors, which could otherwise remain unnoticed. During our tests, we define a manual threshold to cap the number of words to a user-defined threshold. The words considered are sorted according to their frequency as we are interested in relationships between the most commonly used words.

Performance. A fair comparison between the CPU and GPU implementation is not possible since focus was put in optimizing the GPU code. For example, the CPU implementation currently stores the row and column number alongside the actual edge weight in RAM to be able to order the results to the row-major ordering afterwards. To give an order of magnitude, the parallelized Rust CPU implementation (without the subsequent sorting) takes around 12 s (for 10,000 nodes), 42 s (for 20,000 nodes) and 93 s for 30,000 nodes on a 4-core Intel i7-6700 CPU. The implementation is limited to around 35,000 words when $\approx$ 20 GB of RAM are available.

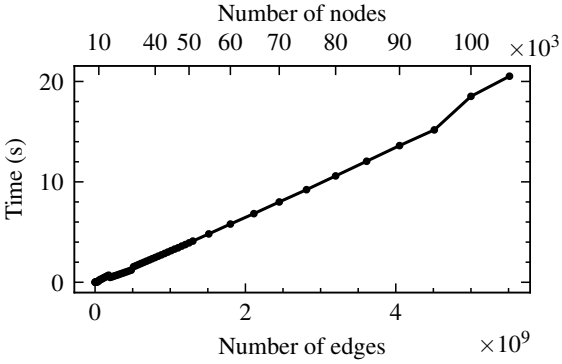

Figure 6: Performance of the GPU code for different number of nodes n. Number of edges via (14).

To test the performance of the GPU code, we measure the kernel execution time (including copying the results back to the host) for a range of number of nodes n in the graph. For every n, we measure the duration 12 times[5] and calculate mean and variance. The results are depicted in Figure 6. The variance is not shown as it is too small to be visible (always less than 1 ms). For 20,000 words, the GPU code takes 484 ms on average, while the CPU implementation needs 42 s. Up to 100,000 nodes (i. e. up to almost 5 billion edges), the GPU implementation takes less than 20 s. Figure 6 also reveals the linear trend of time with increasing number of edges, which was to be expected since a thread is launched for every edge.

Our CUDA implementation is limited by the global memory (6 GB for the GPU at hand). This memory is used to store the resulting edge weights, i. e. one byte per edge. The maximum number of edges we can handle is therefore the available memory divided by 1 byte. To obtain the corresponding number of nodes, we solve (14)

[4] We use the Driver Version 572.42 and CUDA Toolkit 12.8 inside WSL2 (Ubuntu 22.04 jammy).
[5] After every run, the device is re-initialized. Furthermore, we wait 2 s after every run before a new one starts.

for n:

$$n = \frac{1}{2} + \sqrt{\frac{1}{4} + 2 \cdot \text{num edges}} \tag{15}$$

On the GPU at hand, we can handle up to around 107,000 nodes (mean time $\approx 21.3\,\text{s}$) before experiencing "CUDA out of memory errors". Currently we detect the limit, but do not implement a mechanism to go beyond it. One way could to be to detect the error, then copy the results back to the host and continue the computation while shifting the index back to 0. The results are then concatenated on the host. For the further evaluation, we shall content ourselves with the results for the first 100,000 words, which already contain a lot of information of the French language. The binary file holding only the edge weights in row-major order, is 4.66 GB in size.

Graph application. Figure 7 shows the histogram of edge weights. It strongly resembles a normal distribution, which is probably due to how word lengths are distributed in the language. Note that the minimum and maximum achievable alignment score for a word pair depends on the word lengths. To efface this dependency, we normalize every score by dividing by $\max(\text{len}(A), \text{len}(B))$ and then multiply by 100. The resulting histogram is shown in Figure 8. It does not resemble a normal distribution anymore. Most nodes have the smallest score of -100. There is a gap of around width 10 around edge weight 0 that we have no explanation for. The global trend is that many edges have a strong negative edge weight, while a smaller proportion have a strongly positive one.

The binary edge file is converted to a CSV file and imported into the open-source graph visualization software Gephi. As Gephi is far from being able to display 5 billion edges at the same time (let alone import such a file), we have to select specific ranges of edge weights we are interested in. We focus on the positive edge weights as they indicate a higher similarity between words.

Gephi implements various force-directed graph drawing algorithms that can help us gain a better understanding of the data. The principle of these algorithms is that nodes repulse each other, while edges act as springs to pull connected nodes together (taking into account the edge weights). Here, we exclusively use the algorithm by Yifan Hu[6] as we found it to be the most efficient and reliable for our data. Furthermore, we run a modularity analysis using the Louvain method[7] implemented in Gephi. This method is used to detect communities in the graph, i. e. groups of nodes that are more connected to each other than to the rest of the graph. We color the nodes according to the community they belong to (colors don't match between *different* graphs in the following).

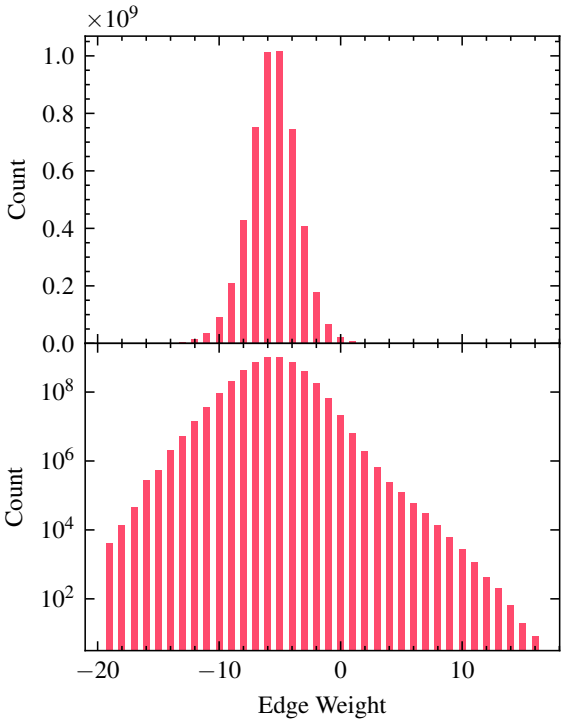

Figure 7: Histogram of edge weights (linear and logarithmic scale). Mean: -1.5, range: $[-19, 16]$.

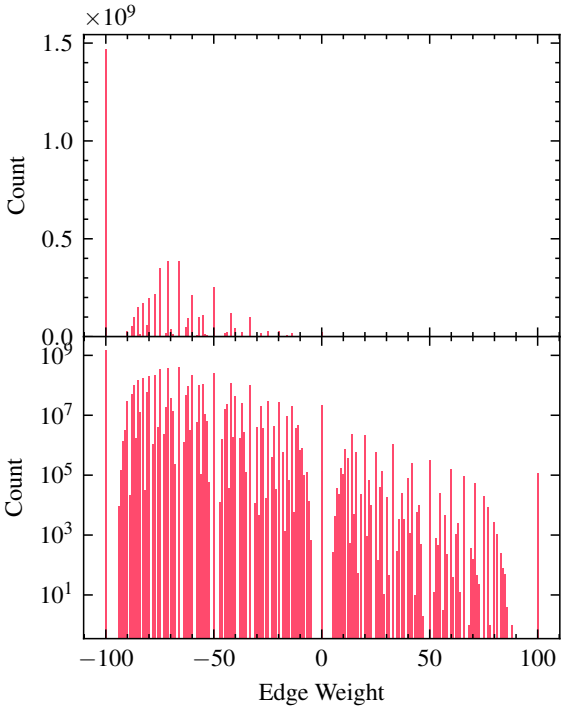

Figure 8: Histogram of normalized edge weights. Linear and logarithmic scale.

[6] Efficient and High Quality Force-Directed Graph Drawing by Yifan Hu in 2006.
[7] Fast unfolding of communities in large networks by Vincent Blondel et al.

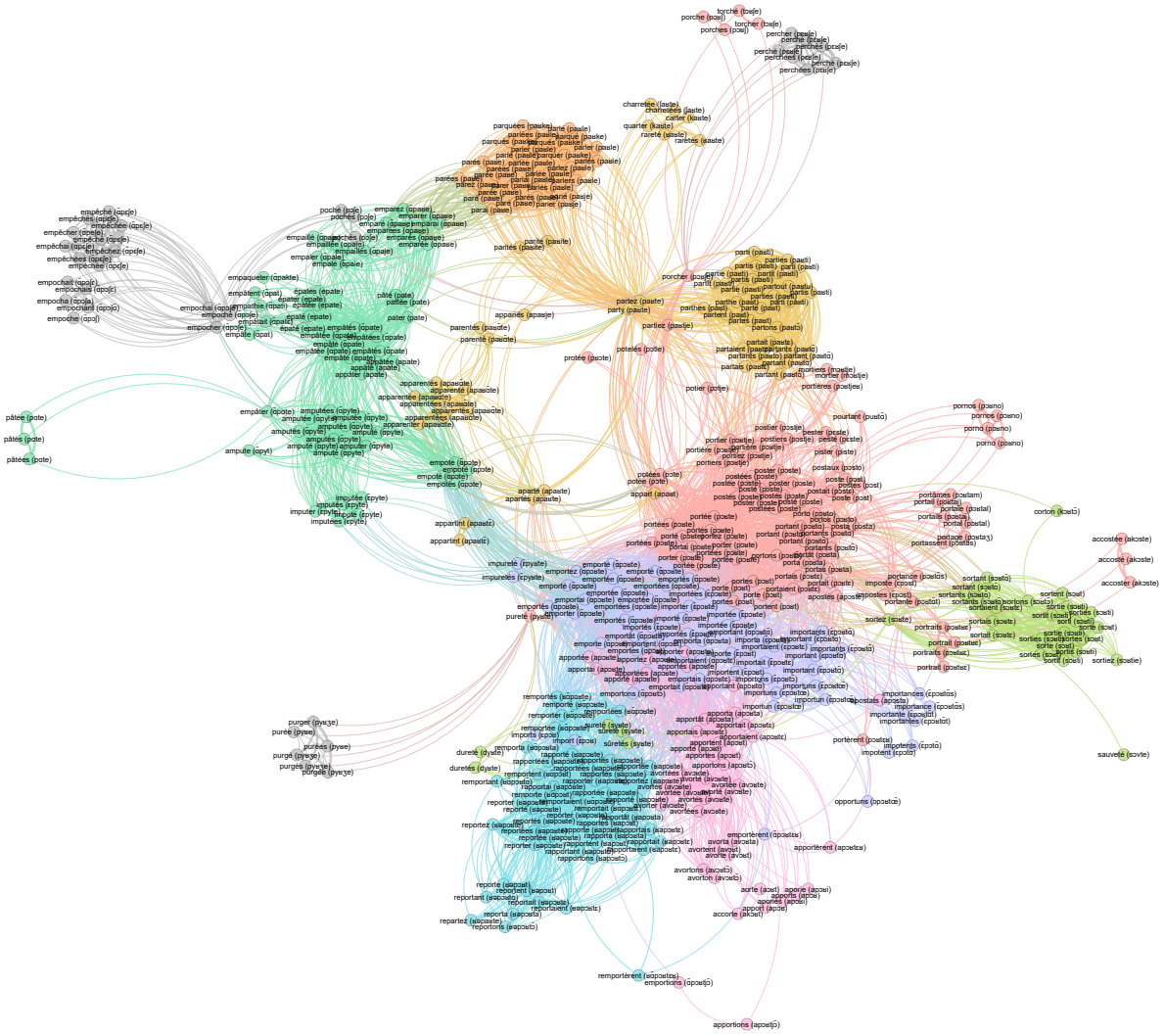

Figure 9: Ego-network (depth 3) of the word *emporter* (to take away) for edge weights in the range $[60, 100]$. This subgraph contains 449 nodes and 5921 edges.

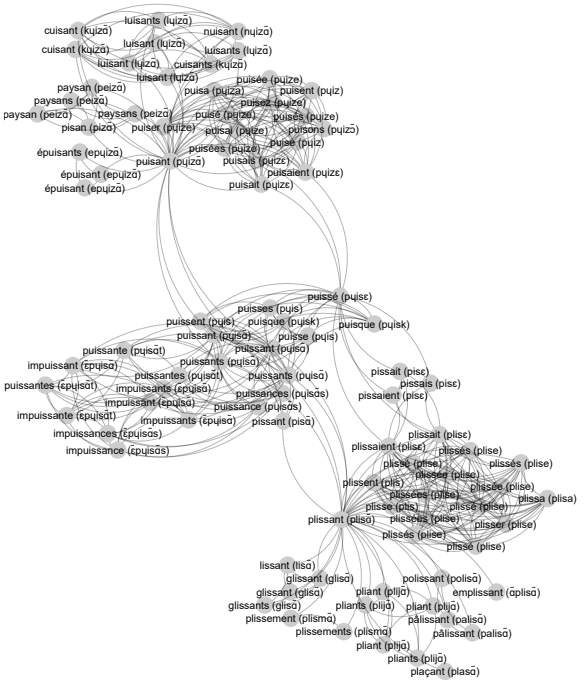

Figure 10: Ego-network (depth 3) of the word *puissant*.

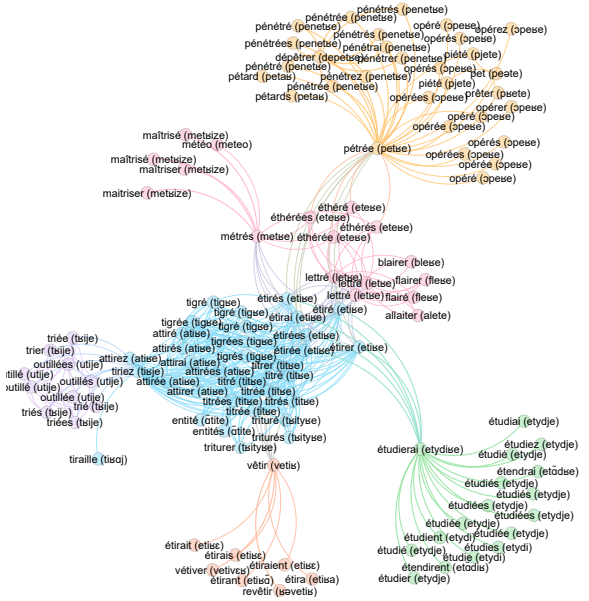

Figure 11: Ego-network (depth 3) of the word *étirer* for edge weights in the range $[40, 49]$.

Some resulting ego-networks are shown in figures 9, 10 and 11. These graphs are constructed by locating neighbors of a word, then finding the neighbors of these neighbors up to a depth of 3. Different slices of edge weights are used, e. g. in Figure 9, we have filtered the graph for edge weights in the range $[60, 100]$ beforehand, which account for a total of 458,529 edges. The networks clearly show that our implementation is working correctly and that Needleman-Wunsch indeed gives a meaningful metric in this context.

- Words with the same pronunciation are as close as possible to each other and reside in the same community. In Figure 11, the words *étudier*, *étudient*, *étudié*, *étudiai*, *étudiées* etc. are close to each other. In Figure 9, inside the red group in the middle-right part, we find different adjective endings for the word *porté* (corresponding to gender and number of the noun it describes, i. e. adjective agreement): porté, portée, portés, portées. We also find the words *porter* and *portez* here with the same prononciation as *porté*. Note that it is claimed that *portai* has the exact same prononciation, which is not correct (the last phoneme is different). This also shows that the dataset itself is not perfect and may contain errors.
- Of greater interest are similar sounding words; as hoped for, they are close to each other in the graph. For example, in Figure 9, we find edges like *emporté* – *porter*, *emporté* – *importé*, *importé* – *impureté* and *impureté* – *pureté*. It is also remarkable that words in one group almost exclusively start with the same letter and correspond to the same word root: *sortir* in the green group on the right, *porter* and *poster* in the red group, *emporter* and *importer* in the violet group, *apporter* in the rose group, *rapporter*, *reporter* and *remporter* in the blue group etc.
- There are stronger and weaker connections between words. In Figure 10, the edge between *puisant*[8] – *épuisant* has weight $^{200}/_{3} \approx 66.7$ (after normalization), while that of *puisant* – *paysans* has the smaller weight $^{300}/_{5} = 60$.

paysans		p	e	i	z	ɑ
puisant		p	ɥ	i	z	ɑ̃
épuisant	e	p	ɥ	i	z	ɑ̃

Table 4: Optimal alignment of three words.

Table 4 shows the corresponding optimal alignments. With a match score of 1, a mismatch score of -1 and a gap penalty $p = -1$, we find a score of 3 (*puisant* – *paysans*) and 4 (*puisant* – *épuisant*). With the normalization discussed on page 7, we indeed obtain the aforementioned edge weights. This example also illustrates that fine-tuning match/mismatch score and gap penalty is important to obtain meaningful results. One might want a greater distance between *paysans* and *puisant* since the /u/ is replaced by the very different sounding /a/. Furthermore, the match/mismatch scores can be adjusted for every pair of phonemes to reflect the subtleties of a language. This can be done by changing the coefficients in the similarity matrix (see line 9 of Algorithm 2.1). In this document, we limit ourselves to the default scoring matrix (1 on the diagonals, -1 elsewhere).

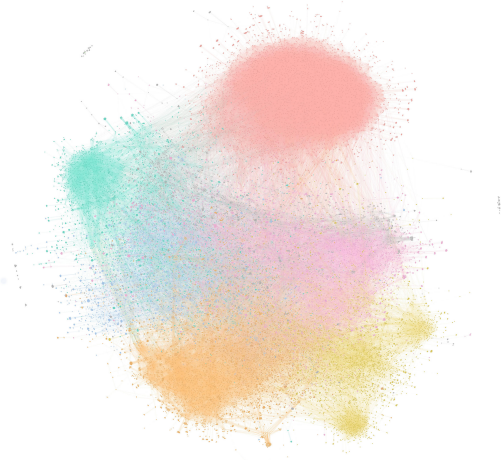

Figure 12: Overview of the graph containing edge weights in the range $[8, 10]$. 30,185 nodes and 306,473 edges are shown.

To broaden the view, we consider a bigger graph in Figure 12, with normalized edge weights 8, 9 or 10. Communities found with Louvain are again color-coded. This very tiny slice of edge weights (compare Figure 8) can already reveal interesting partitions of a language.

- The red community at the top consists almost exclusively of words ending in /jɔ̃/, e. g. *illustration*, *proportions*, *réalisation*, *fluctuation*, *documentation*, *émancipation*, *articulation* etc.
- Below, the rose group encompasses words that end with /œʁ/, e. g. *aspirateur*, *consommateur*, *illustrateur*, *radiateur*. However, this group is less homogeneous and also contains words ending in /ktiv/, e. g. *interactives*, *réparatrice*, *instructif*, *instinctives*.
- The orange group on the bottom mainly comprises words ending in *e*, e. g. *personnalité*, *caractérisé*, *centraliser*, *hospitalité*, *réalité*, *vivacité*, *patienter*, *spécialiser*, *sensibiliser*, *modéliser*.
- In the cyan group on the far left, we find many words that end in ɛl, e. g. *personnel*, *conventionnel*, *professionnel*, *conditionnelles*, *exceptionnelle*, *émotionnels*, *sensationnels*. However, this group is harder to describe since it

[8] Note that this is not missing an additional *s*. This is the word in a phrase like "en *puisant* de l'eau, ...".

also contains different words like *occasionnant*, *collectionneur*, *frictionne*, *crayonné*, *positionnement*.

- The yellow group on the bottom right is easier to classify: it contains words ending in ʒi or ʒik. Among others: *méthodologique*, *topologique* et *topologie*, *holistique*, *diplomatique*, *dramaturgie*, *gastronomie*, *cytologie*, *phénoménologie*, *biologie*, *écologique*, *synchronique*.

We also tried to identify common groups of words *between* two communities in the expectation of finding a "morphing" behavior between them. However, this could not be observed in the graphs and we were not able to find a meaningful way to group these words.

Finally, having translated the problem into a graph structure also allows us to use graph algorithms to discover interesting properties. As an example, Gephi implements the *shortest path algorithm*: users can click on two words and the shortest path between them is calculated and shown in the graph. Beforehand, we filtered the graph to only include the most strong edges. With this, we can find chains like the following (read them aloud to hear the phonetic similarity):

- trottoir → entrevoir → devoir → voire → voile → val → valait → fallait → falaise
- falaise → fallait → palais → passais → dépassait → dépendait → répondait → répond → raison → maison
- confusion → conclusion → exclusion → explosion → exposition → explications → respiration → précipitation → présentation → présenta → présente → présence → présidence → résidence → résistance → existence

6 Conclusion & Outlook

We presented a method to calculate similarity between words based on their phonetic IPA transcription. For this task, we employed the Needleman-Wunsch algorithm with similarity matrix and gap penalty. We implemented this algorithm in Rust and parallelized it on a CPU using the Rayon library and on a consumer Nvidia GPU using the CUDA framework with the *cudarc* Rust library, while writing the kernel itself in C++. We detailed choice of data structures and memory layout to optimize the performance of the GPU implementation. For a graph with 100,000 nodes (words) and almost 5 billion edges (word-pairs), the algorithm takes less than 20 s (including copying back to host). Consistency between the CPU and GPU implementations was verified up to 30,000 nodes (after that, the CPU implementation becomes too slow and consumes too much memory).

Future work could include an adapted version that can read the whole graph of more than 600,000 nodes at the same time by copying back intermediate results from the GPU to the CPU as already outlined in the evaluation. To further speed up the computation, one could consider a more fine-grained parallelization at the level of the score matrix calculation. This is not trivial due to the dependencies between the cells of the matrix. A stencil computation approach might be feasible.

We also demonstrated the practical usability of the resulting edge weights by examining the graphs in Gephi. Even just by looking at small subsections of the full graph, we can deduce interesting properties of the language, e. g. community detection revealed groups with similar word endings and groups with the same root but different endings. Constructing the ego-network of a word allows to find words that are phonetically similar to a given word. One can envision an online platform where this functionality is offered to users to find rhymes or similar sounding words. This can be helpful for language learners as well to foster the playful exploration of a language. Beyond that, future work might tackle the following points.

- Fine-tune the similarity matrix for language subtleties. In this document, we always used a fixed match/mismatch score of 1/-1, while in reality, some phonetic substitutions sound more similar than others. For example, replacing a vowel by a consonant might be more severe than replacing a vowel by another vowel. One might also want to experiment with the gap penalty that was set to -1 throughout.
- So far, we considered a dataset for the French language, while the presented pipeline can work with other languages as well. This can open up the possibility to compare the phonetic structure of different languages and find similarities between them.
- The Louvain algorithm for community detection inherently constructs a multi-level hierarchy of communities. Looking at a lower granularity level (more high-level view) could reveal interesting results. For this purpose, we should consider a parallel Louvain implementation that can work on the whole graph. Limiting oneself to a slice of edge weights (as done here) might bias results or leave interesting structures unnoticed.
- The size of the dataset can be reduced by merging words with the same lexeme, e. g. for words *remportés*, *remportant*, *remportée*, *remportées* etc. we could only consider the lexeme *remporter*. This could help gain a better and more general understanding of the phonetic structure of the language as the graphs would be clearer.

Glossary

IPA International Phonetic Alphabet. 1, 2, 4, 10

POS Part-of-Speech. 4

PTX Parallel Thread Execution. 5