PC-2017/18 Edit Distance using CUDA

Mansueto Pierluigi
E-mail address
pierluigi.mansueto@stud.unifi.it

Giuntini Andrea
E-mail address
andrea.giuntini@stud.unifi.it

1. Introduction

In computational linguistic and computer science, edit distance is a way of quantifying how dissimilar two strings are to one another by counting the minimum number of operations required to transform one string into the other one. This algorithm is used in many important fields: the most popular examples are natural language processing, where automatic spelling correction can determinate the right corrections for a misspelled word by selecting words from a dictionary that have a low distance to the word in question, and bioinformatics, where edit distance is used to quantify the similarity between two sequences of DNA.

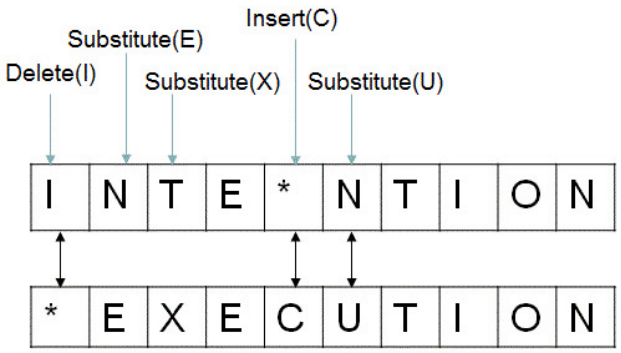

Figura 1. Edit Distance example

Many approaches are implemented to quantify the Edit Distance between words. We dealt with one of this: Levenshtein distance. This one consider only three operations: copy, insert and delete of a letter.

However, the weakness of this algorithm is that it suffers from high computational complexity. Indeed, its use is not recommended when you want to run this method in practical large-scale settings. For example, in bioinformatics, where you must quantify how much two sequences of DNA, composed by millions of nucleotides (characters) in length, are similar, the usage of this algorithm takes too much computational time. In order to delete this drawback, during the years, some techniques have been implemented to obtain edit distance between words with a decrease of the computational time of the algorithm, using long words too. These approaches used the parallelism capabilities of the Graphics Processing Unit (GPU).

We have implemented the main approach to accelerate the Levenshtein edit distance computation using the capabilities of GPU and we did a comparison, based on SpeedUp, between this parallel approach of the edit distance and the serial one.

2. How to represent edit distance

There are many ways to obtain the edit distance between two words: many data structures have been used during the years. The most classic method is the integer matrix, where, if you consider two words of length M and N, has a number of $M+1$ rows and $N+1$ columns. This data structure is the most intuitive and easy to manipulate. To obtain the edit distance between two words, one of length M and N respectively, we calculated the value in the cell of the matrix denoted by the M-th row and the N-th column. To calculate this value, we had to compute the other cells of the matrix, following this behavior:

$$H_{i,j} = \min \begin{cases} H_{i-1,j-1} + Score \\ H_{i,j-1} + 1 \\ H_{i-1,j} + 1 \end{cases} \quad (1)$$

where the $Score$ value in our implementation is 0 if the character of the first sequence matches the character of the second sequence; otherwise, the $Score$ value is 1. In this data structure, we can see the weakness of the edit distance. Indeed, if you want to find edit distance between two words of length M and N,the algorithm will have a complexity of $O(MN)$, because you must calculate the values in all cells of the matrix.

After this overview, it's even more obvious why we implemented a parallel approach of this algorithm to try to reduce its computational time.

3. CUDA

CUDA is a general purpose programming model on NVIDIA GPUs. It enables explicit GPU memory management through its libraries and compiler directives. In particular, CUDA C is an extension of standard ANSI C with an handful of language extensions to enable heterogeneous programming (the creation of an application that use both CPU and GPU), and also straightforward APIs to manage devices, memory, and other tasks. In this heterogeneous programming, CPU is in charge of computing the sequential part of the program and copy data from its memory to GPU memory, calling the CUDA function (Kernel) to perform program-specific computation in GPU. After a kernel function call from CPU, GPU must compute the numerically intensive part of the code. Finally, when GPU finishes, a copy of the data must be done from GPU memory to CPU memory, in order to obtain the result of the program.

The GPU is a SIMT model (Single Instruction Multiple Threads): the Kernel function is implemented as a sequential code but each instruction of the Kernel is computed by many execution units at the same time, which are the threads of the GPU. These ones are grouped in blocks that are grouped in a grid in the GPU too: in the next paragraphs you will see how to choose the number of threads in a block and the number of blocks in a grid in our project.

Why did we choose the GPU for a parallel approach instead of CPU? Because in the CPU we dealt with a few tens of threads, that are generally heavyweight entities, while in GPU we dealt with tens of thousands of threads per GPU, generally extremely lightweight. In addition, if the GPU must wait on one group of threads, it simply begins executing another one.

For the reasons explained in this description, you will see in the paragraph *Experimental result* how, in the comparison between the serial approach and the parallel one, SpeedUp increases when the words which you want to compare have a larger length.

4. Parallelization of edit distance

To obtain edit distance between two words, we had to compute the value of the cells of the matrix already described. As shown in the formula in the paragraph *How to represent edit distance*, if you want to find value in a specific cell you must already have three values: one of the cell to the left of the specific one, one of the cell above the specific one and one of the cell at the top left of the specific one. So, the value to find depends on three cells. For this reason, the computation of these four cells must not be done in a parallel approach together. There is another way to compute cells of the matrix: a diagonal way. In this way, we considered only the cells in one of the skew diagionals of the matrix to be computed in the parallel approach one by one.

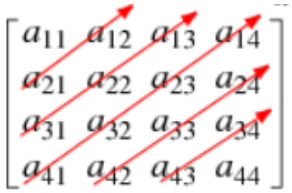

Figura 2. Diagonal Implementation

In this way, a cell in a skew diagonal can be computed in parallel with the other ones, because the three values on which it depends have been already computed. In this matrix there are $M + N - 1$ skew diagonals: CPU must call the kernel function $M + N - 1$ times, so that in each call GPU must compute in parallel the cells of a specific skew diagonal. This is the idea to implement a parallel approach of edit distance: the dependencies between cells is cut off using a diagonal implementation of the algorithm, and GPU computes each cell of a skew diagonal in a parallel way.

4.1. Host code

The host code is the one which is computed by CPU. First, CPU allocates the matrix to find edit distance between two words and initializes it. The initialization is done in this way:

$$\forall i \in [0, M] \quad H_{i,0} = i \tag{2}$$

$$\forall j \in [0, N] \quad H_{0,j} = j \tag{3}$$

Then, CPU must copy the matrix and strings, related to words, in the form of array of char, from its memory to GPU global memory (memory shared by all blocks of threads in the grid). Furthermore, CPU must tell to GPU which skew diagional have to be computed and how many blocks and threads per block must be used. For the first task, a number between 0 and $M + N - 1$ that indicates which skew diagonal must be computed, is sent to the GPU as a parameter. For the second task, we have chosen 256 threads for each block. In terms of SpeedUp, the difference between 256 threads and other values is irrelevant, so we decided to use this number because we thought that it led to the best performances. The number of blocks depends on the number of cells to be computed in a kernel function call.

$$numB = \lceil \frac{skewDiagonal - incRows - incCols}{numT} \rceil \tag{4}$$

We decided to introduce the variables $incRows$ and $incCols$ because they are needed to get over the problem of the matrix boundaries. With the formula $(skewDiagonal - incRows - incCols)$ we get how many cells are allocated in a specific skew diagonal. The, when each cell of

any skew diagonal is computed by GPU, CPU receives the matrix from GPU global memory.

4.2. Device code

Device code is the one computed by GPU. Each thread must process, in a parallel way, a single cell of the skew diagonal related to the kernel function. This cell is denoted by two index:

$$indexC = threadIdx.x + blockIdx.x * blockDim.x \quad (5)$$

$$indexR = skewDiagonal - incRows - indexC \quad (6)$$

In this way, Thread 0 of GPU must process the cell denoted by the biggest number of row and the smallest number of column. Thread 1 must process the cell to the top right of the one that Thread 0 must process and so on. Depending on the number of cells to compute, it may happens that the number of threads available is bigger than the number of cells. For this reason, we decided to make a control at the beginning of the device code:

$$1 + incRows \leq indexC + 1 + incRows \quad (7)$$

$$indexC + 1 + incRows \leq skewDiagonal - incCols \quad (8)$$

so that each thread, starting from Thread 0 (this is why we added $(1 + incRows)$ to the control), computes a cell if this latter is not out of the matrix boundaries ($(slice - incCols)$). After the control, a thread calculates the value of the cell, related to it, as seen in the formula in the paragraph *How to represent edit distance*. To decide if a character of a string is equal to one of a second string we used a function created by us called `StringToDecimal`. This function returns a number between 1-26 (26 is the number of alphabet letters) from each letter (for example at the letter A correspond the number 0, at the letter B the number 1 and so on . . .). Another important fact: we converted the matrix from a 2D array to a 1D array due to the lack of support for large-scale 2D arrays.

5. Experimental Results

In the tests of the application, we studied the behavior of SpeedUp comparing a serial approach of edit distance to a parallel one, where we used the entire capabilities of GPU.

$$S_P = \frac{t_S}{t_P} \quad (9)$$

Computer specifics:

- Intel Core i7-7700HQ 2.80GHz Octa-core
- 16 GB RAM
- 250 GB HDD
- GeForce GTX 1050 4 GB

The behavior is studied by varying the length of the words involved, considering words characterized by the same length. For each words size, 5 different tests have been developed, and the averages are reported in the following table.

Words size	CPU Time	GPU Time	Speed Up
50	28,6	30,022	0,9526347345
100	96,4	74,136	1,3003129384
250	544,6	176,094	3,0926664168
500	2435,6	364,728	6,677853085
1000	3556	714,976	4,9735935192
2000	17750,6	1431,778	12,3975923642
3000	48061	2095,3	22,9375268458
4000	95137,6	2853,364	33,3422584711
5000	149259,4	3688,246	40,4689383517
6000	224212,6	4508,458	49,731549013
7000	320455,6	5250,64	61,0317218472
8000	447298,2	6245,86	71,6151498753
15000	1,82E+06	12355,926	146,9066745787
31000	1,47E+07	29145,05	505,1554209034

SpeedUp grows markedly when the length of words increases. However, if the words are characterized by a small length, then you will notice that the sequential approach is more convenient than the parallel one. But, while the length of words increases, the parallel approach becomes more and more convenient than the sequential one.

After this experimental results, GPU turns out to be one of the best candidate to implement a parallel approach of edit distance.

Riferimenti bibliografici

[1] M. Bertini, *10_gpu_cuda_1.pdf*, Università degli Studi di Firenze, 2017-2018

[2] M. Bertini, *11_gpu_cuda_2.pdf*, Università degli Studi di Firenze, 2017-2018

[3] K. Balhaf, M. A. Shehab, W. T. Al-Sarayrah, M. Al-Ayyoub, M. Al-Saleh, Y. Jararweh, *Using GPUs to Speed-Up Levenshtein Edit Distance Computation*, Jordan University of Science and Technology, Irbid, Jordan, 2016-2017