PC-2022/23 final: Parallel edit distance

Carlo Ceccherelli
E-mail address
carlo.ceccherelli@stud.unifi.it

Abstract

In this project we will see two version of a program that computes the edit distance between two strings: a sequential version and a parallel version; the latter one will be implemented both in OpenMP and CUDA. We will compare the performance of the two versions showing the speedup that can be reached using a parallel computing approach.

Future Distribution Permission

1. Introduction

The edit distance (or Levenshtein distance) between two strings is defined as the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other [1]; it follows a recursive relation showed in 1 where $\operatorname{tail}(x)$ is the string x without its first character.

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

This recursive relation can be transformed in a dynamic programming algorithm, known as the Wagner-Fischer algorithm, showed in 1. Let m and n be the lengths of the two strings to be compared, this algorithm fills an $(m+1)\times(n+1)$ matrix D and returns $D[m][n]$ as the edit distance; as we can see in the pseudocode, the first row and firs column of D are initialized, then every matrix element is computed comparing only the elements on its top (deletion), left (insertion) and top-left (copy or substitution).

Algorithm 1

```
Require: $a, b$ : strings to be compared
  $m = a.length$
  $n = b.length$
  for $i = 0$ to $m$ do
    $D[i][0] = i$
  end for
  for $j = 1$ to $n$ do
    $D[0][j] = j$
  end for
  for $i = 1$ to $m$ do
    for $j = 1$ to $n$ do
      if $a[i] \neq b[j]$ then
        $D[i][j] = 1 +$
                $\min D[i-1][j-1], D[i-1][j], D[i][j-1]$
      else
        $D[i][j] = D[i-1][j-1]$
      end if
    end for
  end for
  return $D[m][n]$
```

1.1. Main idea

The approach showed in algorithm 1 is clearly sequential and presents data dependencies because every computation of a matrix element requires informations from previous iterations. To parallelize this algorithm the standard way is to consider the *antidiagonals* of the matrix: if we adjust the two nested loops in 1 to iterate over antidiagonals (outer loop) and their elements (inner

Listing 1 random generation of a string

```
string generateRandString(int size)
{
    const int ch_MAX = 4;
    char alpha[ch_MAX] = {'a','b','c','d'};
    string result = "";
    for (int i = 0; i < size; i++)
        result += alpha[rand() % ch_MAX];
    return result;
}
```

loop) we can see, as shown in Fig 1, that the elements of an antidiagonal can be fully computed in parallel because each of them depend on the previous two antidiagonals (already computed). So while we have to mantain the order of execution in the outer loop (each antidiagonal needs the previous two to be computed) we can execute the inner loop in parallel.

$$\begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix}$$

Figure 1. The main antidiagonal of a matrix

2. Implementation

We will now see the implementation details of both sequential and parallel versions of the program, following the considerations presented in the introduction. As mentioned in the abstract, i decided to implement two parallel version, one using OpenMP and one using CUDA; this allows us to compare also CPU and GPU programming approaches.

2.1. Sequential version

In the sequential version i simply implemented the algorithm 1, using an utility function `generateRandStrings(size)` (showed in listing 1) to create the strings to be compared. The implementation is presented in listing 2.

Listing 2 sequential edit distance

```
int size = 50000;
string A = generateRandString(size);
string B = generateRandString(size);
uint lenA = A.size();
uint lenB = B.size();

uint** D = new uint*[lenA+1];
for (int i = 0; i < lenA+1; i++)
    D[i] = new uint[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] = 1 + min({D[i][j-1],
 ↪  D[i-1][j], D[i-1][j-1]});
        else
            D[i][j] = D[i-1][j-1];
    }
}
cout << "edit distance: " << D[lenA][lenB];
```

2.2. Parallel version with OpenMP

As mentioned in the introduction, the main idea behind the parallelization of the Wagner-Fischer algorithm is to iterate over *antidiagonals* of the matrix; so the outer loop will iterate over these antidiagonals d and the inner loop over its elements i; also an index j is needed to indicate the element of the second string to be compared with the *i-th* element of the first. I decided to parallelize the inner loop with *tasking*, in particular exploiting the directives `#pragma omp master` and `#pragma omp taskloop`, which together will generate and execute, for every iteration of the outer loop, as many tasks as the number of iterations of the inner loop.

With this said, if we only adjust the two loops to scan the dynamic programming matrix by its antidiagonals, the parallel algorithm performs much worse than the sequential one; this is caused by an irregular memory access pattern: in fact in the parallel algorithm the matrix D is stored as a linear array (so in row major order) but its elements are not accessed in a ”linear” way (while in the sequential version the access pattern follows contiguous memory locations) so the cpu cannot leverage optimizations like *caching* or *RAM burst*. Having acknowledged this downfall, i thought of two approaches to solve the issue. The first approach uses three arrays as *diagonal fronts* to compute every element of the dynamic programming matrix; while this method solves the memory locality problem, it brings out another downfall due to poor workload assignement for each thread, in fact a thread is responsible of computing only a single element of the current antidiagonal. With this said the second approach arises, combining the idea of iterations over antidiagonals with the concept of *tiling*: in particular we can divide the matrix in tiles and make the outer loop iterate over antidiagonal tiles; in this case each thread of the inner loop would take a tile as a work unit instead of a single element. Doing so the threads would still run in parallel, because the tiles in an antidiagonal ”stripe” are independent of each other, and they would have an adequate work load to justify threading costs. The first problem regarding memory locality would not be an issue in this approach due to the fact that each thread compute the elements of its tile sequentially, following the Wagner-Fischer algorithm presented in 1. This last approach, using tiling, is showed in listing 3. As already said, each thread has the duty to compute a tile of the matrix, this is done via the `computeTile()` function (showed in listing 4) which, starting from the tile coordinates, identifies the starting indexes of the submatrix it has to work on, and then procedes to fill this submatrix with the Wagner-Fischer algorithm.

2.3. Parallel version with CUDA

CUDA (Compute Unified Device Architecture) is a general purpose programming model available on NVIDIA GPUs. It permits parallel programming on GPUs exploiting their architechtures. As opposed to OpenMP, CUDA programming provides a much higher number of

Listing 3 parallel edit distance: tiling approach

```
int tilesA = ceil((float)(lenA)/tileWidth);
int tilesB = ceil((float)(lenB)/tileWidth);
#pragma omp parallel default(none)
↪   shared(D, A, B, lenA, lenB, tilesA,
↪   tilesB, tileWidth)
{
    int dmin = 1-tilesA;
    int dmax = tilesB;
    #pragma omp master
    {
        for(int d = dmin; d < dmax; d++)
        {
            int imin = max(0, d);
            int imax = min(tilesA + d,
↪   tilesB);
            #pragma omp taskloop
            for(int i = imin; i < imax;
↪   i++)
            {
                int j = tilesA + d - i - 1;
                computeTile(i, j, lenA,
↪   lenB, A, B, D, tileWidth);
            }
        }
    }
}
cout<<"parallel edit distance:
↪   "<<D[lenB*(lenA+1) + lenA];
```

”lightweight” threads, organized in a grid-block fashion represented in Fig 2. CUDA distinguishes the *host code*, executed on the cpu, from the *device code*, executed on the gpu; functions running on the gpu are called *kernels* and when they're called they spawn the grid-block thread organization.

To parallelize the Wagner-Fischer algorithm using CUDA, the main concept of computing the dynamic programming matrix by antidiagonals remains; however it turns out that simply adapting the tiling approach from the OpenMP version is not the best choice, this is due to the architechtural differences between CPUs and GPUs; in fact the many GPU threads are more suitable to execute a narrow set of tasks. With this said, an approach that assignes each antidiagonal element to a thread yields the best performance. This method is similar to the first

Listing 4

```
void computeTile(int I, int J, uint lenA,
↪   uint lenB, string A, string B, uint* D,
↪   int tileWidth)
{
    I = I * tileWidth + 1;
    J = J * tileWidth + 1;
    for(int i = I; i < lenB+1 && i <
↪   I+tileWidth; i++)
    {
        for(int j = J; j < lenA+1 && j <
↪   J+tileWidth; j++)
        {
            if(A[j - 1] != B[i - 1])
                D[i * (lenA+1) + j] = 1 +
↪   min({D[i * (lenA+1) + j - 1], D[(i - 1)
↪   * (lenA+1) + j], D[(i - 1) * (lenA+1) +
↪   j - 1]});
            else
                D[i * (lenA+1) + j] = D[(i
↪   - 1) * (lenA+1) + j - 1];
        }
    }
}
```

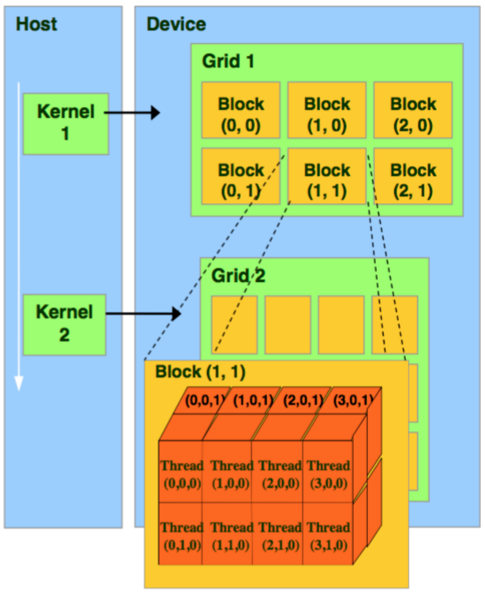

Figure 2. Speedup of the parallel version

approach mentioned in 2.2, it's based on an outer loop iterating on the antidiagonals of the dynamic programming matrix and three arrays representing the three antidiagonals needed at each iteration (meaning at each single antidiagonal computation); to make sure i mantained in memory only these three arrays during the whole execution, i had to declare them with the size of the largest antidiagonal. This approach is showed in listing 5: we can see the antidiagonals initialization on the host, followed by the declaration of the device antidiagonals and three pointers useful for fast swapping between the three arrays; next we have allocation and copy of host arrays to device memory and block and grid size definition. Once everything is set up, start and end indexes of outer loop are computed (`dmin` and `dmax`), these values are designed so as to have the main antidiagonal with index `d=0`. Finally the loop is executed calling at every iteration the `editDistKernel()` which parallely computes the element of a single antidiagonal; after each kernel call the antidiagonal fronts are advanced, making the second-last antidiagonal of the previous iteration the last of the current and so on. Note that there is no need for synchronization at system level (i.e. `cudaDeviceSynchronize()`) because every kernel call (and also the call to `cudaMemcpy()` after the loop) is executed on the default stream, so while it's true that kernel calls are asynchronous with respect to host code, subsequent kernel invocations will execute one after the other, ensuring to work with the right antidiagonals after they had been correctly advanced.

The kernel code is showed in listing 6 where we can see how each thread controls if it's inside the boundaries of the current antidiagonal and procedes to compare string chars and previous antidiagonals elements to compute its value. Note how, if we are computing an antidiagonal with index $d < 1$ i.e. an antidiagonal before the main one, the first and last value are computed a priori because they correspond respectively to elements on the first row and first column of the dynamic programming matrix.

Listing 5 CUDA version

```
int* currDiag = new int[lenA+1];
int* prevDiag = new int[lenA+1];
int* prevprevDiag = new int[lenA+1];
prevprevDiag[0] = 0;
prevDiag[0] = 1;
prevDiag[1] = 1;

int* d_currDiag;
int* d_prevDiag;
int* d_prevprevDiag;
int* d_currDiagPtr;
int* d_prevDiagPtr;
int* d_prevprevDiagPtr;

cudaMalloc((void**)&d_currDiag,
 ↪   (lenA+1)*sizeof(int));
cudaMalloc((void**)&d_prevDiag,
 ↪   (lenA+1)*sizeof(int));
cudaMalloc((void**)&d_prevprevDiag,
 ↪   (lenA+1)*sizeof(int));
cudaMemcpy((void*)d_prevprevDiag,
 ↪   (void*)prevprevDiag,
 ↪   (lenA+1)*sizeof(int),
 ↪   cudaMemcpyHostToDevice);
cudaMemcpy((void*)d_prevDiag,
 ↪   (void*)prevDiag, (lenA+1)*sizeof(int),
 ↪   cudaMemcpyHostToDevice);
//pointers asignment for fast swapping
d_prevprevDiagPtr = d_prevprevDiag;
d_prevDiagPtr = d_prevDiag;
d_currDiagPtr = d_currDiag;

int bDim = 128;
int gDim = ceil((float)(lenA+1)/bDim);
int dmin = 2-lenA;
int dmax = lenB+1;
int ed;
for(int d = dmin; d < dmax; d++)
{
    editDistKernel<<<gDim, bDim>>>(devA,
↪   devB, lenA, lenB, d_prevprevDiagPtr,
↪   d_prevDiagPtr, d_currDiagPtr, d);

    int* tmp = d_prevprevDiagPtr;
    d_prevprevDiagPtr = d_prevDiagPtr;
    d_prevDiagPtr = d_currDiagPtr;
    d_currDiagPtr = tmp;
}
cudaMemcpy((void*)&ed,
 ↪   (void*)&d_prevDiag[lenA],
 ↪   1*sizeof(int), cudaMemcpyDeviceToHost);
cout << "edit distance: " << ed;
```

Listing 6 CUDA kernel

```
__global__ void editDistKernel(char* devA,
 ↪   char* devB, int lenA, int lenB,
 ↪   unsigned int* d_prevprevDiagPtr,
 ↪   unsigned int* d_prevDiagPtr, unsigned
 ↪   int* d_currDiagPtr, int d)
{
    int tid =
 ↪  blockIdx.x*blockDim.x+threadIdx.x;
    int j = lenA+d-tid;
    if(d < 1)
    {
        if(tid == 0)
            d_currDiagPtr[0] = MIN(lenA+d,
 ↪  lenB+1);
        else if(tid == lenA+d)
            d_currDiagPtr[lenA+d] =
 ↪  MIN(lenA+d, lenB+1);
    }
    if(tid < MIN(lenA+d, lenB+1) && tid >
 ↪  MAX(0, d-1))
    {
        if(devA[j-1] != devB[tid-1])
            d_currDiagPtr[tid] = 1 +
 ↪  MIN(d_prevDiagPtr[tid],
 ↪  MIN(d_prevDiagPtr[tid-1],
 ↪  d_prevprevDiagPtr[tid-1]));
        else
            d_currDiagPtr[tid] =
 ↪  d_prevprevDiagPtr[tid-1];
    }
}
```

3. Tests and results

Finally we will see how the two versions perform and how much speedup the parallel version can bring. Tests are executed on a laptop with 16GB RAM and an i5-1135G7 quad core processor clocked at 2.4GHz. Time measurements in the code are done using `chrono::high_resolution_clock` and include the portion of code relative to instantiation/initialization and computation of matrix/antidiagonal fronts (generation of the two random strings to compare is not included); the code is measured 10 times for each configuration (number of threads, strings dimension) to evaluate the average execution time. In table 1 are reported execution times of sequential and parallel versions for OpenMP on different string sizes with 4 and 8 threads, in Fig 3 is plotted the speedup reached with the OpenMP version. Similarly in table 2 are reported execution times for the CUDA version varying both strings sizes and number of thread per block, in Fig 4 is plotted the speedup reached with the CUDA version.

Speedup is evaluated by the formula

$$S = \frac{t_s}{t_p} \tag{2}$$

where t_s indicates the execution time of the sequential version and t_p the execution time of the parallel version.

string chars	seq time	par time (4 thr)	par time (8 thr)
10000	0.38s	0.15s	0.13s
30000	3.51s	1.30s	1.11s
50000	10.12s	4.05s	3.41s

Table 1. Execution times of sequential and parallel version for OpenMP

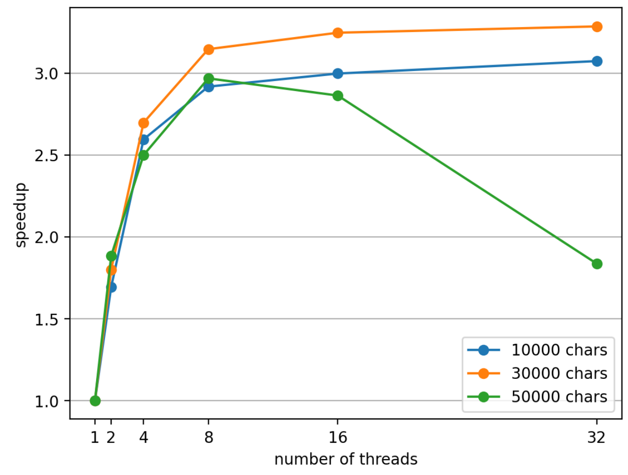

Figure 3. Speedup of OpenMP version

block size / string size	128	256	512	1024
10000	0.1268s	0.1266s	0.1256s	0.1282s
30000	0.3911s	0.3921s	0.3995s	0.4440s
50000	0.7934s	0.8035s	0.8165s	0.9123s
100000	2.3260s	2.3487s	2.3903s	2.9213s
150000	4.8100s	4.8657s	5.0153s	6.3399s

Table 2. Execution times of parallel version for CUDA

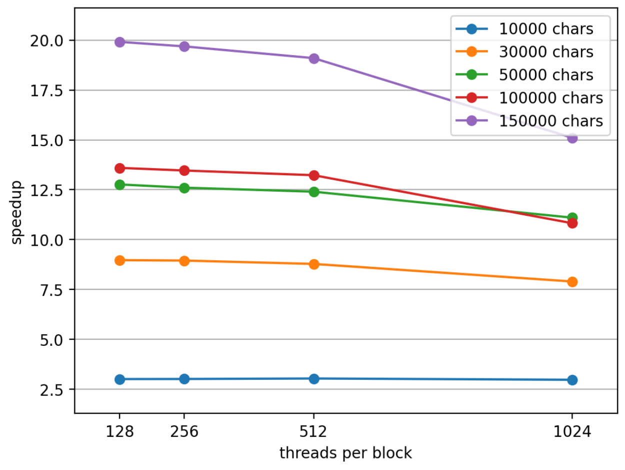

Figure 4. Speedup of CUDA version

References

[1] Wikipedia contributors. Levenshtein distance — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=1147354410`, 2023. [Online; accessed 8-April-2023].