Parallel edit distance

Final project for Parallel Computing course

Carlo Ceccherelli

Università degli Studi di Firenze

Table of contents

Introduction

Goal

Write a program that computes the edit distance between two strings.

Edit distance (Levenshtein)

the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other

Implement two versions:

- Sequential
- Parallel using **OpenMP**
- Parallel using **CUDA**

Analyse execution times and *speedup* reachable when applying parallel programming.

Problem definition

Recursive relation where tail(x) is the string x without its first character

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

Can be transformed into a dynamic programming algorithm (Wagner-Fischer)

- let m and n be the a and b string lengths
- fills an $(m+1) \times (n+1)$ matrix D
- returns $D[m][n]$ as the edit distance

Wagner-Fischer algorithm

```
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 $D[m][n]$
```

Main idea

Every matrix element to be computed depends on its top, left and top-left element

- the W-F algorithm presents data dependencies

Consider antidiagonals of the matrix D

- elements of an antidiagonal are independent of each other!
- each antidiagonal needs the previous two to be computed

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

Implementation (OpenMP)

Sequential version

Implements the Wagner-Fischer algorithm

Uses an utility function to generate random strings to be compared

```
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;
}
```

Parallel version with OpenMP

OpenMP: a shared memory API for parallel programming.
Includes a set of directives to instruct the compiler on how to parallelize the program.

Main idea: adjust the loops in W-F algorithm so that:

- outer loop iterates over antidiagonals d (with main andiagonal having index 0)
- inner loop iterates over its elements i

Compute also an index j to identify the char of the 2nd string to be compared with the i-th char of the 1st.

Parallel version with OpenMP - Problem

Only adjusting the two loops makes the parallel version run extremely slower than the sequential one

- caused by poor memory locality
- cpu can't exploit optimizations like **caching** and **RAM burst**

Solution: **tiling**

- iterate over antidiagonals made of *tiles*
- assign each thread to a tile on an antidiagonal
- compute the tile sequentially with the W-F algorithm so as to mantain memory locality

Use openmp **tasking** to parallelize inner loop

- `pragma omp master`
- `pragma omp taskloop`

Parallel version with OpenMP - Tiling

```
int tilesA = ceil((float)(lenA)/TW);
int tilesB = ceil((float)(lenB)/TW);
#pragma omp parallel shared(D,A,B,lenA,lenB,tilesA,tilesB){
    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);
            }
        }
    }
}
cout<<"parallel edit distance: "<<D[lenB*(lenA+1) + lenA];
```

Parallel version with OpenMP - Tiling

```
void computeTile(int I, int J, int lenA, int lenB, string A,
↪   string B, int* D){
    I = I * TW + 1;
    J = J * TW + 1;
    for(int i = I; i < lenB+1 && i < I+TW; i++){
        for(int j = J; j < lenA+1 && j < J+TW; 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];
        }
    }
}
```

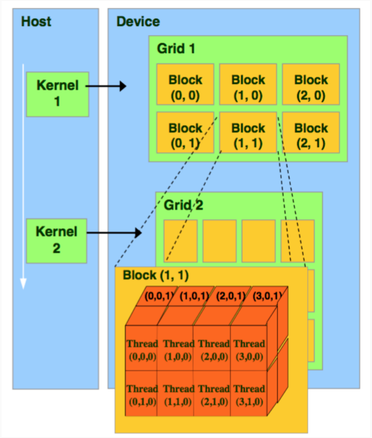
Implementation (CUDA)

Parallel version with CUDA

CUDA: parallel programming model that exploits GPU architechtures.

Differences with CPU parallelism:

- distinction between **host** (cpu) and **device** (gpu)
- much more "lightweight" threads organized in **grid-block** fashion



Parallel version with CUDA

Adapting the tiling approach seen in OpenMP does not yield best performance

- gpu threads work better when assigned to single/small task

Shift to a **antidiagonal fronts** approach:

- assigns each antidiagonal element to a thread
- instantiate three arrays representing second-last, last and current antidiagonal of each iteration

  ```
  int* currDiag = new int[lenA+1];
  int* prevDiag = new int[lenA+1];
  int* prevprevDiag = new int[lenA+1];
  ```
- call a **kernel** for each antidiagonal to be parallely computed
- advance the three antidiagonals after each kernel invocation

Parallel version with CUDA

Declare device arrays and relative pointers for fast swapping.
Allocate and copy arrays to device memory

```
int* d_currDiag, d_currDiagPtr;
int* d_prevDiag, d_prevDiagPtr;
int* d_prevprevDiag, d_prevprevDiagPtr;
cudaMalloc();
...
cudaMemcpy();
...
d_prevprevDiagPtr = d_prevprevDiag;
d_prevDiagPtr = d_prevDiag;
d_currDiagPtr = d_currDiag;
```

Parallel version with CUDA

```
//kernel parameters definition
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);
    //advance antidiagonal fronts
    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;
```

Parallel version with CUDA

Edit distance kernel: antidiagonals before main one need to have first and last element set beforehand.

```
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];
}
```

Tests and results

Tests setup

Tests executed on:

- laptop with 16GB RAM and an Intel i5-1135G7 quad core processor clocked at 2.4GHz for **OpenMP version**
- server with 64GB RAM, Intel Xeon Silver 4314 16-core processor clocked at 2.4GHz and an NVIDIA RTX A2000 graphics card

Time measurements are done using `chrono::high_resolution_clock`

- execution times are measured 10 for each test and then averaged

Speedup is evaluated as $S = \frac{t_s}{t_p}$ where t_s indicates the execution time of the sequential version and t_p the execution time of the parallel version

Time measurements

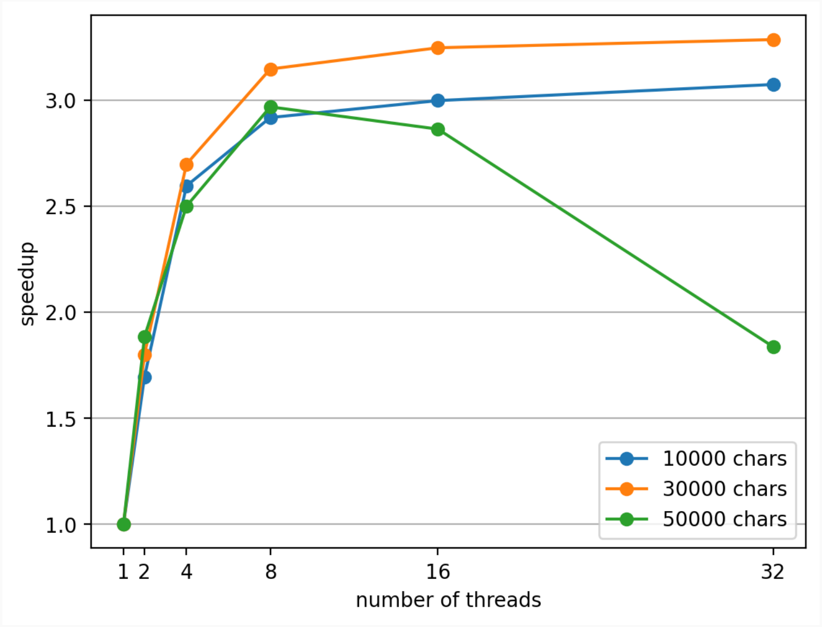
OpenMP

string size	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

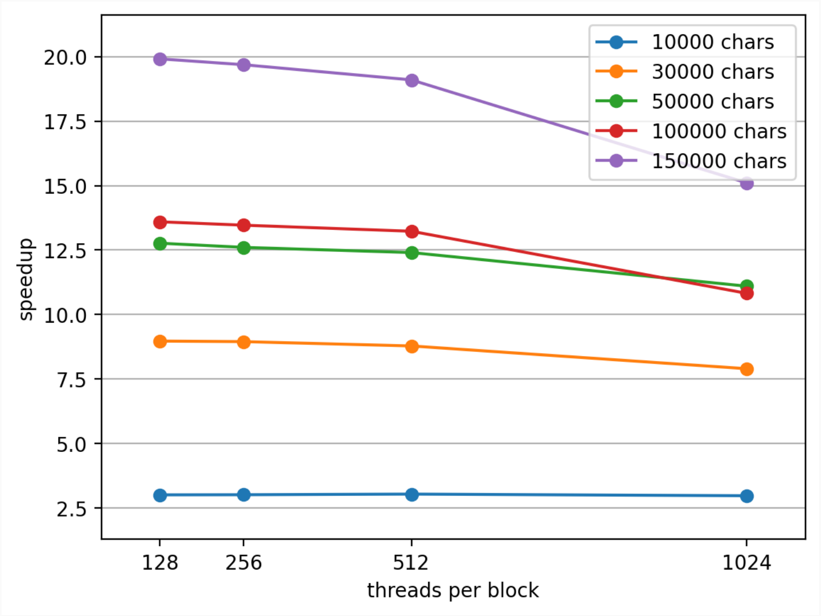
CUDA

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

Speedup - OpenMP version



Speedup - CUDA version



Thank you for your attention!