INF560 Project : Approximate Pattern Matching

Riade Benbaki, Ali Mammadov

January-Mars 2021

1 Introduction

Pattern matching is a classic and recurrent problem in which a pattern is searched for in another text. It is used in various applications, like spell checkers, spam filters, plagiarism detection, bioinformatics, information retrieval systems and etc. In this project, we worked on a pattern matching algorithm based on the **Levenshtein Distance**. It is a string metric for measuring the difference between two sequences. The Levenshtein distance between two words is the minimum number of one-character edits (insertions, deletions, or replacements) required to convert one word to another. The main objective of this work is the parallelization the previous matching algorithm because in order to be able to handle larger file sizes.

2 Explanation of the parallelism Approaches/Choices

2.1 Overview

The goal of the project is to mix MPI, OpenMP and CUDA in order to obtain maximum performance. In order to determine which parts to parallelyze with which tool and paradigm, we start by analyzing the sequential version. The sequential version start by reading the file and the patterns we are searching for. Then, for each pattern, it runs a loop over every starting position in the file and computes the Levenshtein distance between the corresponding strings. Computing this distance can not be parallelized as it uses a dynamic programming approach in which we need previous results in order to compute the next step. For each pattern, the work done for each starting position is independent from the other one, because we are only interseted in knowing whether we have a match or not.

2.2 MPI

We divide the file over all available MPI ranks evenly (not exactly since one rank, 0 in our case, needs to handle an extra part). Each MPI rank computes the number of matches between each pattern and a string of size $file\_size/MPI_{size}$. The results are then added together to obtain the final result. In this part, an important detail is that each rank needs to receive not only his own part of the data, but also some shadow cells at the end. These cells belong to the part of the next rank, but are necessary for the current rank as well in order to process the last starting position in it's own part.

Let buf be a buffer containing the entire file, Let n_bytes be the the number of characters in the file (and it's size as well since a char is encoded in one bit), $size$ be the number of MPI ranks, and $rank$ the rank of the current MPI process. Then each MPI process $rank > 0$ receives a string $local\_buf$ starting from $buf[(rank-1)*(n\_bytes/size)]$ up to

$$buf[min(rank*(n\_bytes/size)+max\_pat-1, n\_bytes)],$$

where max_pat is the size of the biggest pattern which we compute when reading the patterns. We take the minimum in order to not exceed the end of buf is the case where the maximum pattern size if bigger the size of the next chunk. The case where $rank = 0$ is much simpler since this rank processes the last chunk and therefore does not need shadow cells.

2.3 OpenMP

In every MPI rank, we run a loop through all starting values in the chunk size of the current rank. Iterations in this loop are independent, since they do not modify the data even if different

iterations can process overlapping strings. For this reason, we can use OpenMP threads to parralilize this loop and perform a reduction in order to get the total number of matches found by all threads.

2.4 CUDA

In order to get the most of CUDA, we need to give the GPU the simplest tasks. In our case, we can not parralelize on a deeper level than the one already used in OpenMP. For this reason, we chose to use CUDA and OpenMP on the same level, each of them working, in parallel, on a part of the chunk size of the current MPI process.

CUDA threads are limited by the available memory in the GPU. Each CUDA thread (and OpenMP) needs access to the text string (shared), to an array *column* of the same size as the pattern (private to each thread), and to an array of size the number of threads in which every thread will write its results. Moreover, if multiple MPI process use the same GPU, the available memory needs to be divided over all the ranks using this GPU. To compute the maximum number of CUDA threads each MPI rank can use, we use a pessimistic approeach in which we assume that all MPI ranks use the same GPU and divide the memory accordingly.

Let nth_{max} be the maximum number of CUDA threads allocated to each process. It needs to verify :

$$[nth_{max}(max\_pat + 2) \times sizeof(int) + buf\_size \times sizeof(char)]\, size \leq freeGpUMemory()$$

Once nth_{max} is computed from the previous formula, each MPI process divides the workload between a maximum number of nth_{max} CUDA threads, and the available OpenMP threads.

3 Parallel Algorithm

Algorithm 1 Parallel Algorithm

Result: Number of the approximate patterns
Init:
 Allocate tables;
 Send tables to devices;
 for *each p in length patterns* **do**
  send pattern to devices
   kernelCall();
   #pragma omp parrallel
   OpenMp initialization;
   #pragma omp for reduction ompMatches
   for *each j in OpenMp part* **do**
    ompMatches += (levenshtein() < aproxFactor) ;
  end
  #pragma omp single
   Synchronize kernel -> finalcudaCall(results);

  #pragma omp for reduction cudaMatches
   cudaMatches = sum(results);

  matches[p] = cudaMatches + ompMatches;
   free(some memory);

end
MpiReduce(matches[p]);
 free(some memory);

The parallel algorithm (see Algorithm 1) starts with reading the patterns (every MPI process reads all the patterns). Then Process_0 reads the file then distributes it equally between all the processes as explained in the previous section. Then each process is assigned to the corresponding GPU device in Round_Robin fashion and calculates the maximum number of CUDA threads as explained in 2.4.

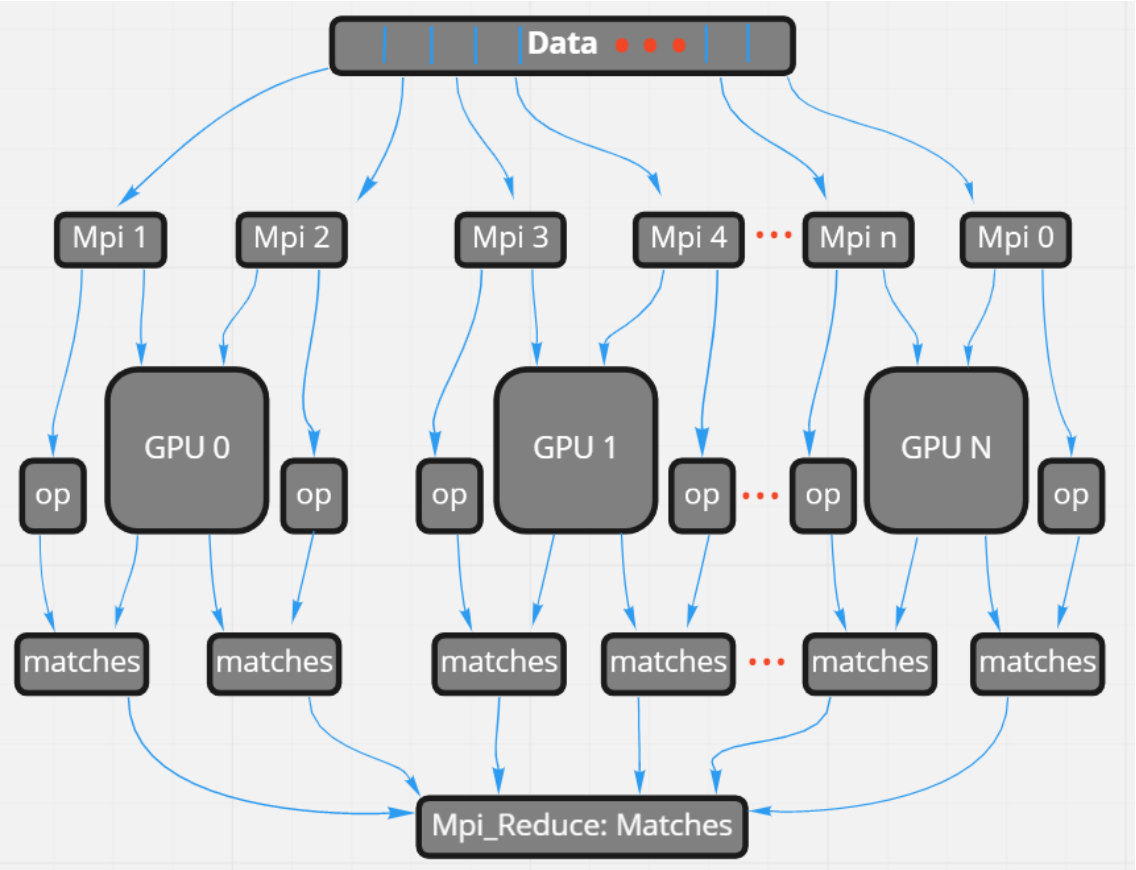


Figure 1: Architecture of the Algorithm

Next, each process divides its part of the file between CUDA and OpenMP using a user provided **ratio** then allocates and sends the text buffer, the current pattern, a table *column* necessary for computations, and a results table where every thread stores the number of matches found in the GPU and calls the CUDA kernel function. The kernel function call is asynchronous, so the process launches the kernel then enters the OpenMP parallel zone without waiting for the CUDA call to finish. Here, OpenMP threads process their part of the file using a guided for OpemMP construct. The first thread that ends its job synchronizes CUDA's kernel function and copies the result back from the GPU to the host. These results are then summed up by OpenMP threads to give the total matches found by CUDA. The results of all threads are then summed up, and the number of matches found by all threads is sent to Mpi_reduce. All allocated memory is cleared as well. See Fig. 1 for a visual explanation of the main idea behind the parallelization.

4 Experimental Evaluations

4.1 CUDA Workload ratio

An important parameter of our program is the CUDA workload ratio, which controls how much of the data is processed by CUDA , the rest being process by OpenMP. It is clear that the optimal ratio would depend on the number of CUDA and OpenMP threads, as well as on the total size of text considered. The optimal ratio would be the one where CUDA and OpenMP finish their respective work at the same time. However, when CUDA and OpenMP finish, OpenMP threads are then used to combine all of CUDA's results. This step takes therefore a longer time if the workload and the number of CUDA threads are bigger. To visualise how the optimal ratio changes in respect to all these parameters, we run experiments where we vary the ratio in the interval $[0, 1]$ and compute the average runtime in different configurations. All the following figures are obtained with one MPI rank.

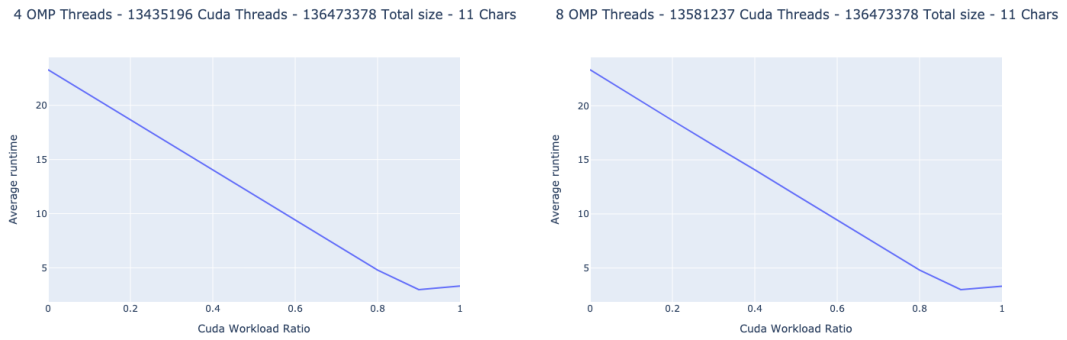


Figure 2: Average runtime with 1.3 $\times 10^7$ available CUDA Threads

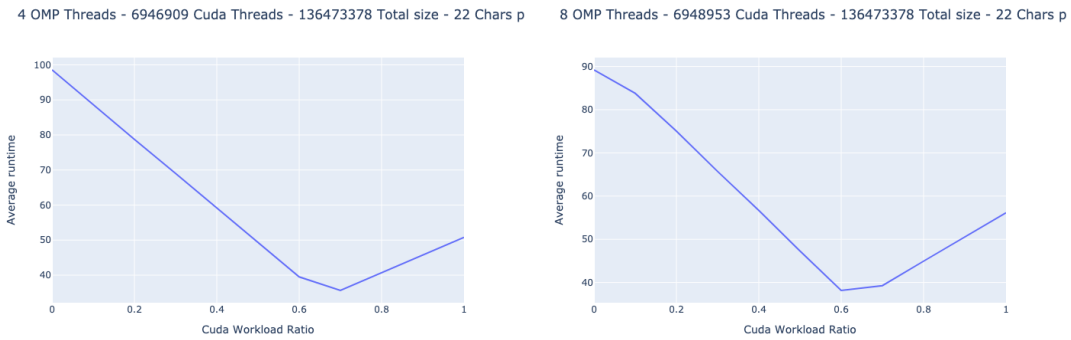
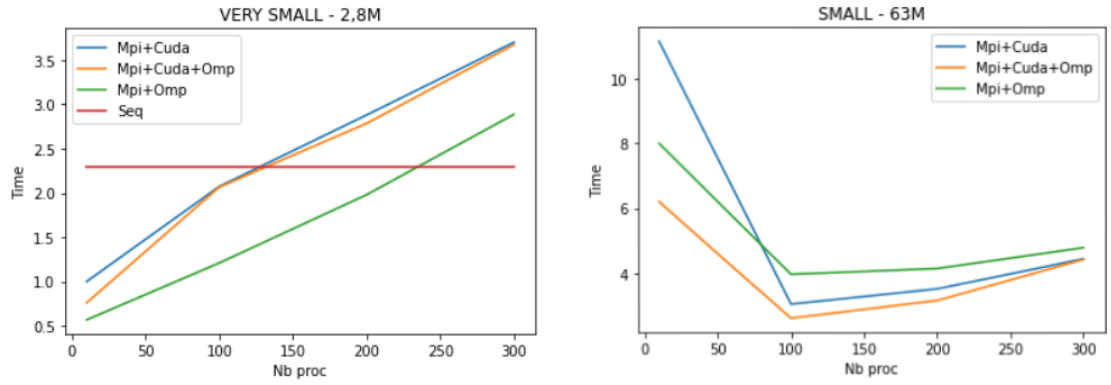


Figure 3: Average runtime with 6.9 $\times 10^6$ available CUDA Threads

4.2 Speedup Evaluations

Figures 4 - 7, illustrate results of experiments on 7 different files with various sizes from very small to very large. The Y axis represents the time of transfer and computations on the file, while the X axis represents the number of MPI processes(the number of GPUs is 10 times less than the processes). The speedup is the ratio between the sequential calculation time and the parallel calculation and transfer time.

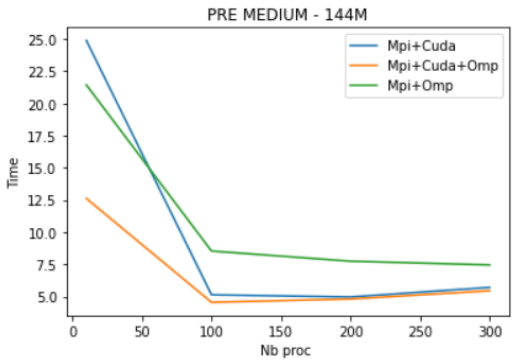
$$Speedup = \frac{Sequential\ calculation\ time}{Parallel\ calculation\ time + transfer\ time}$$



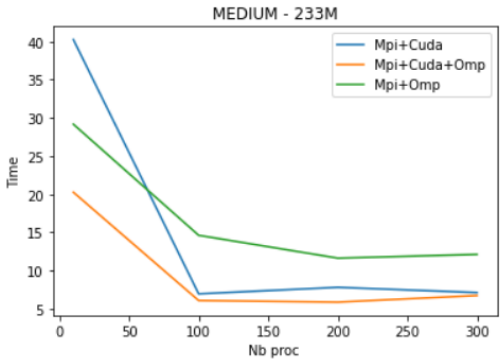

```
Minimum speedup = 0.6186212674082705
Maximum speedup = 4.046322719955613
```

```
Minimum speedup = 3.570232421317228
Maximum speedup = 15.331004418483946
```

Figure 4: Experiments on small data

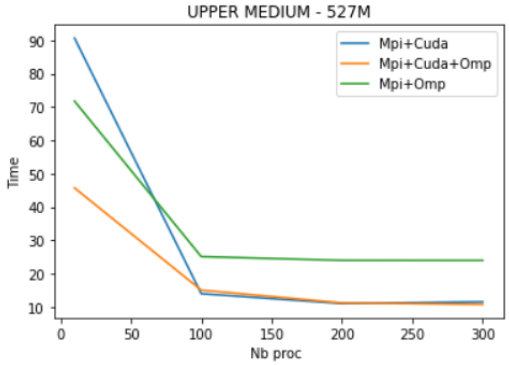



```
Minimum speedup = 5.39116870652277
Maximum speedup = 29.437689435296004
```

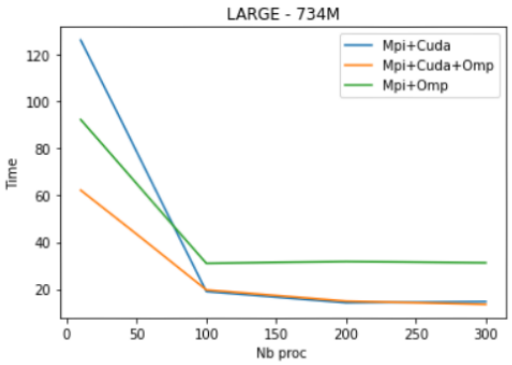



```
Minimum speedup = 3.6988748998460608
Maximum speedup = 25.33639914612596
```

Figure 5: Experiments on medium data

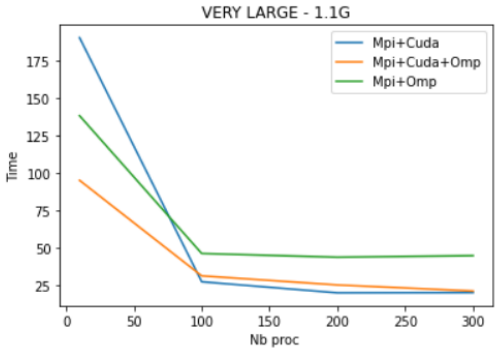



```
Minimum speedup = 4.571884137171331
Maximum speedup = 38.61584246698981
```




```
Minimum speedup = 3.732405832628397
Maximum speedup = 34.67503064440221
```

Figure 6: Experiments on large data




```
Minimum speedup = 3.665600413736744
Maximum speedup = 34.27204043887857
```

Figure 7: Experiments on very large data

From the experiments we see that, for large files, we could achieve very high speedups (up to 38) but not so much for smaller files. So roughly we can say that it is better to use the full parallelization for files larger than the medium size. For smaller ones actually it is not so important to activate Cuda, as we see Mpi+OpenMP can manage them. And finally for very small files there is no need to any parallelization.

5 Content of Source Code

The source code of the project mainly consists of 2 source files: apm_ompic.cu and apm_ompic.c. The **apm_ompic.cu** file contains the CUDA calls and consists of several functions:

- *compMatches()* - the most important. It is the CUDA kernel function that does all of the necessary computations including the Levenshtein function for every thread.
- *kernelCall* - this function is used for calling the kernel function from MPI.
- *finalcudaCall* - function for synchronizing the kernel function. It's called by one of the OpenMP threads when it finished it's own computations.
- *cuda_malloc_cp* - it first allocates the memory for the given size then copies the given buffer to the allocated memory on the GPU.
- *cuda_malloc* - a wrapper around cudaMalloc, returns a pointer to allocated memory in GPU.
- *cuda_free* - a wrapper aroud cudaFree.
- *AssignDevices* - assigns the processes to the devices in a Round-Robin approach.
- *checkGpuMem* - prints information about free and total device memory. Only used for debugging.
- *freeMem* - returns free space in device memory.

The **apm_ompic.c** file consist of several functions of sequential version and the main function which includes parallel algorithm that described in the next sections.

6 Conclusion

This project consisted in parallelizing an Approximate Pattern Matching algorithm based on calculation of the Levenshtein distance. The parallelization was done by combining MPI, CUDA and OpenMP. Using the three models introduces several synchronization challenges and parameters that influence the performance. The CUDA workload ratio is an example of such parameters for which we were not able to find a universal law allowing the computation of the optimal ratio.