Polytech Paris-UPMC

Merge and sort project in CUDA

Presentation

7th December 2017

Alexandre Do
Driss Aloui
MAIN5

Mr. Abbas Turki Lokman
Mr. Moustafa Sali

N9-IPA PARALLELISME AVANCE

2017 - 2018

The steps of our work :

- Understanding the merge and sort method in GPU
- Implementation of the partitioning
- Implementation of the merge and sort
- Application on a dataset
- Performance analysis of our implementation
- How can we improve our program ?

Understanding the merge and sort methods in GPU

GPU Merge Path - A GPU Merging Algorithm

Oded Green *
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA 30332
ogreen@gatech.edu

Robert McColl
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA 30332
rmccoll3@gatech.edu

David A. Bader
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA 30332
bader@cc.gatech.edu

ABSTRACT

Graphics Processing Units (GPUs) have become ideal candidates for the development of fine-grain parallel algorithms as the number of processing elements per GPU increases. In addition to the increase in cores per system, new memory hierarchies and increased bandwidth have been developed that allow for significant performance improvement when computation is performed using certain types of memory access patterns.

Merging two sorted arrays is a useful primitive and is a basic building block for numerous applications such as joining database queries, merging adjacency lists in graphs, and set intersection. An efficient parallel merging algorithm partitions the sorted input arrays into sets of non-overlapping sub-arrays that can be independently merged on multiple cores. For optimal performance, the partitioning should be done in parallel and should divide the input arrays such that each core receives an equal size of data to merge.

In this paper, we present an algorithm that partitions the workload equally amongst the GPU Streaming Multiprocessors (SM). Following this, we show how each SM performs a parallel merge and how to divide the work so that all the GPU's Streaming Processors (SP) are utilized. All stages in this algorithm are parallel. The new algorithm demonstrates good utilization of the GPU memory hierarchy. This approach demonstrates an average of 20X and 50X speedup over a sequential merge on the x86 platform for integer and floating point, respectively. Our implementation is 10X faster than the fast parallel merge supplied in the CUDA Thrust library.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

Keywords

Parallel algorithms, Parallel systems, Graphics processors, Measurement of multiple-processor systems

*Corresponding author

ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.

1. INTRODUCTION

The merging of two sorted arrays into a single sorted array is straightforward in sequential computing, but presents challenges when performed in parallel. Since merging is a common primitive in larger applications, improving performance through parallel computing approaches can provide benefit to existing codes used in a variety of disciplines. Given two sorted arrays A,B of length $|A|,|B|$ respectively, the output of the merge is a third array C such that C contains the union of elements of A and B, is sorted, and $|C| = |A| + |B|$. The computational time of this algorithm on a single core is $O(|C|)$ [2]. As of now, it will be assume that $|C| = n$.

The increase in the number of cores in modern computing systems presents an opportunity to improve performance through clever parallel algorithm design; however, there are numerous challenges that need to be addressed for a parallel algorithm to achieve optimal performance. These challenges include evenly partitioning the workload for effective load balancing, reducing the need for synchronization mechanisms, and minimizing the number of redundant operations caused by the parallelization. We present an algorithm that meets all these challenges and more for GPU systems.

The remainder of the paper is organized as follows: In this section we present a brief introduction to GPU systems, merging, and sorting. In particular, we present Merge Path [8, 7]. Section 2 introduces our new GPU merging algorithm, GPU Merge Path, and explains the different granularities of parallelism present in the algorithm. In section 3, we show empirical results of the new algorithm on two different GPU architectures and improved performance over existing algorithms on GPU and x86. Section 4 offers concluding remarks.

1.1 Introduction on GPU

Graphics Processing Units (GPUs) have become a popular platform for parallel computation in recent years following the introduction of programmable graphics architectures like NVIDIA's Compute Unified Device Architecture (CUDA) [6] that allow for easy utilization of the cards for purposes other than graphics rendering. For the sake brevity, we present only a short introduction to the CUDA architecture.

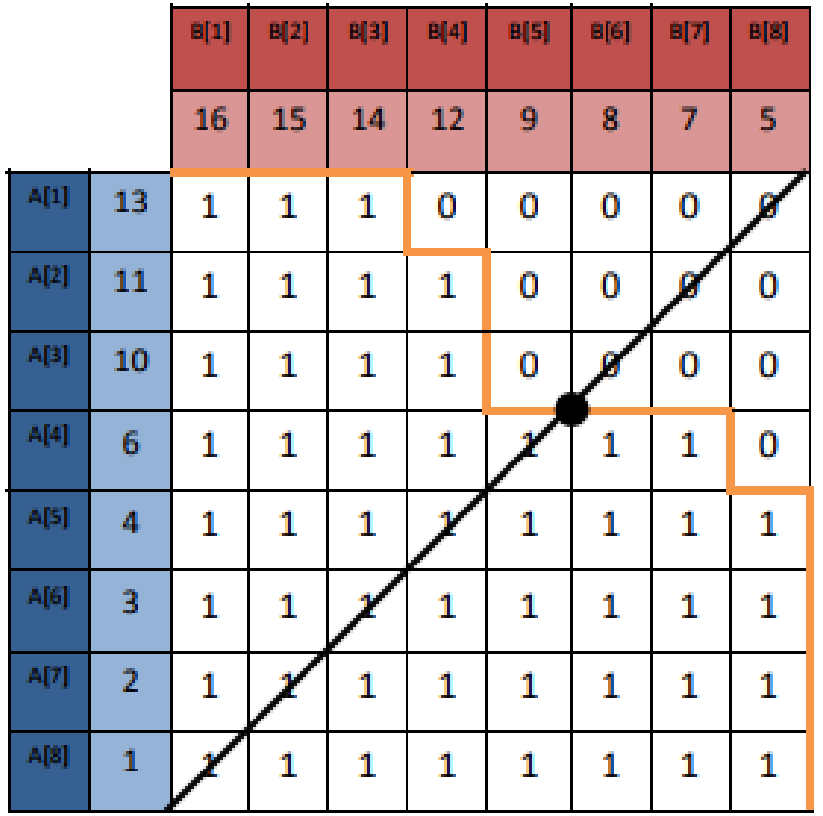
To the authors' knowledge, the parallel merge in the Thrust

GPU Merge Path - A GPU Merging Algorithm

Oded Green - Robert McColl - David A. Bader

MergePath concept

- Input: 2 sorted arrays
- Orange path = comparison decisions
  => merged output array
- Black point = median of output array



		B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
		16	15	14	12	9	8	7	5
A[1]	13	1	1	1	0	0	0	0	0
A[2]	11	1	1	1	1	0	0	0	0
A[3]	10	1	1	1	1	0	0	0	0
A[4]	6	1	1	1	1	1	1	1	0
A[5]	4	1	1	1	1	1	1	1	1
A[6]	3	1	1	1	1	1	1	1	1
A[7]	2	1	1	1	1	1	1	1	1
A[8]	1	1	1	1	1	1	1	1	1

Implementation of the partitioning

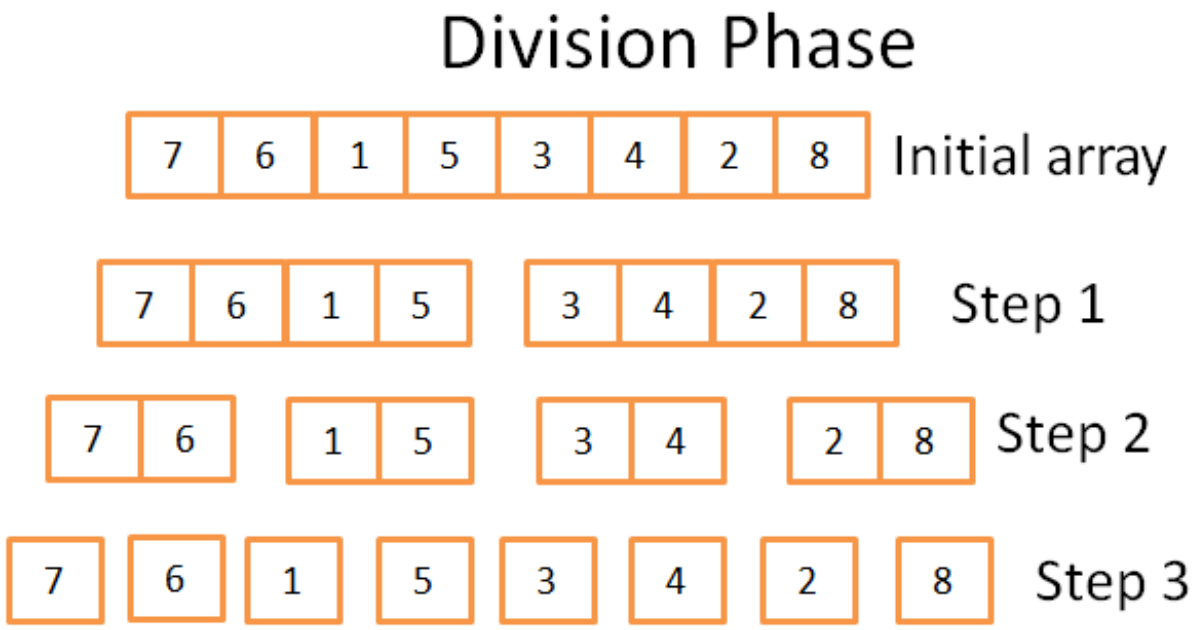
Followed the algorithm (binary search)

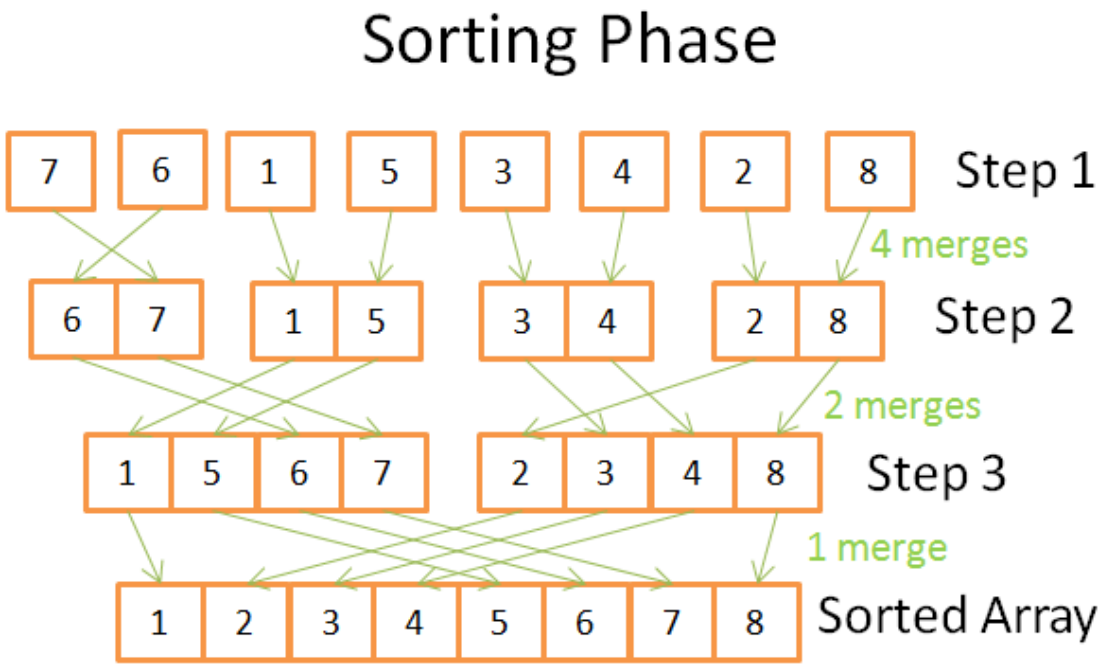
Algorithm 1 Pseudo code for parallel Merge Path algorithm with an emphasis on the partitioning stage.

```
A_diag[threads] ⇐ A_length
B_diag[threads] ⇐ B_length
for each i in threads in parallel do
  index ⇐ i * (A_length + B_length) / threads
  a_top ⇐ index > A_length ? A_length : index
  b_top ⇐ index > A_length ? index − A_length : 0
  a_bottom ⇐ b_top
  // binary search for diagonal intersections
  while true do
    offset ⇐ (a_top − a_bottom)/2
    a_i ⇐ a_top − offset
    b_i ⇐ b_top + offset
    if A[a_i] > B[b_i − 1] then
      if A[a_i − 1] ≤ B[b_i] then
        A_diag[i] ⇐ a_i
        B_diag[i] ⇐ b_i
        break
      else
        a_top ⇐ a_i − 1
        b_top ⇐ b_i + 1
      end if
    else
      a_bottom ⇐ a_i + 1
    end if
  end while
end for
for each i in threads in parallel do
  merge(A, A_diag[i], B, B_diag[i], C, i * length/threads)
end for
```

Implementation of the merge and sort

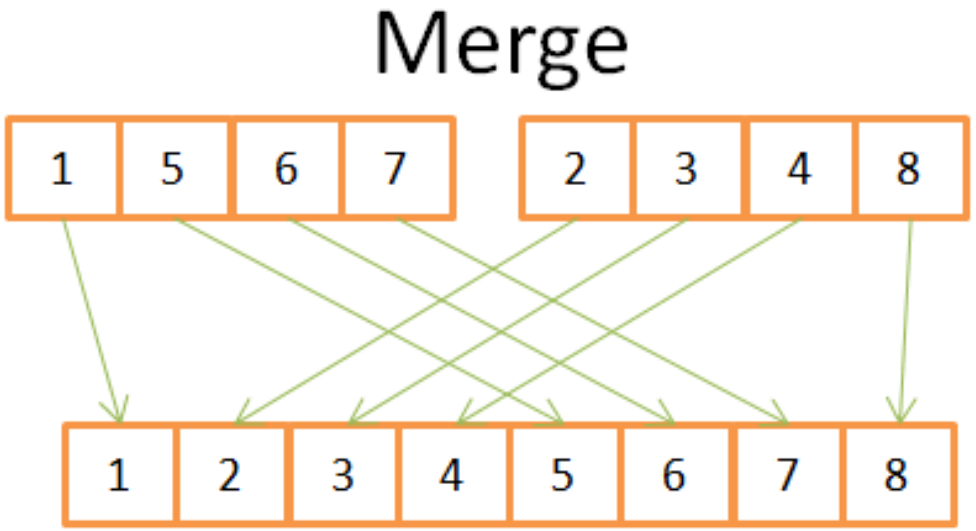
- Implemented sorting phase in sequential





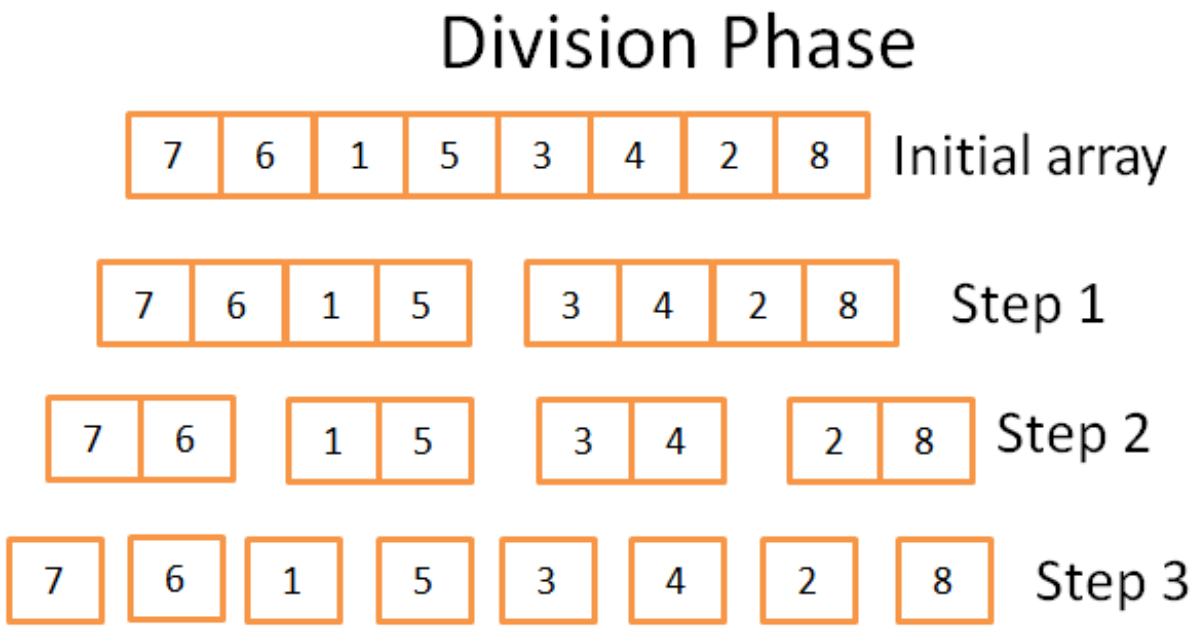
Implementation of the merge

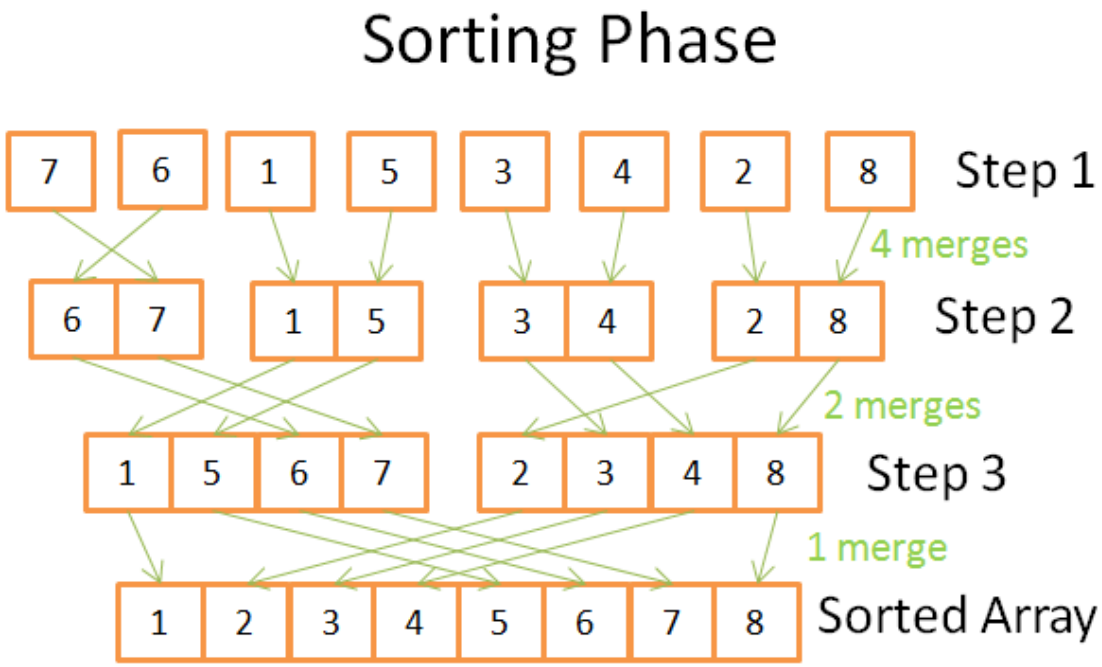
- Merge two sorted arrays



Implementation of the merge and sort

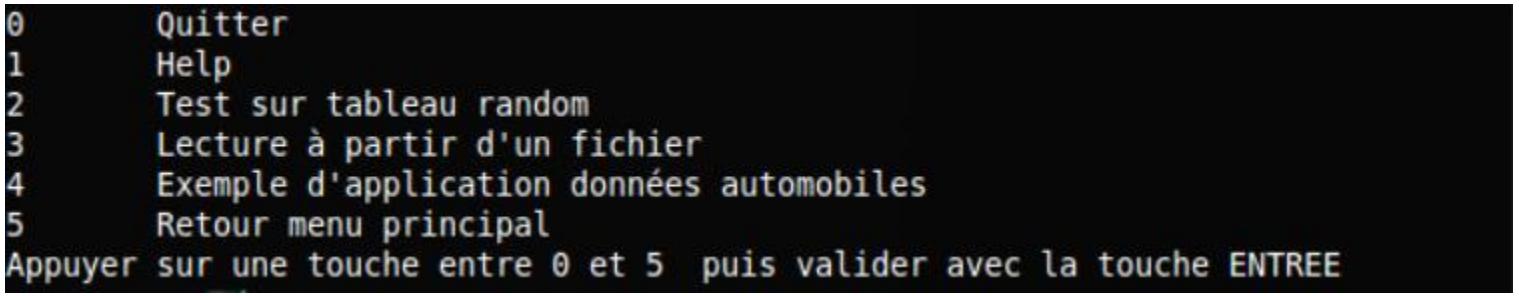
- Adapted in GPU with partitioning





Application on a dataset

- Created an interactive menu

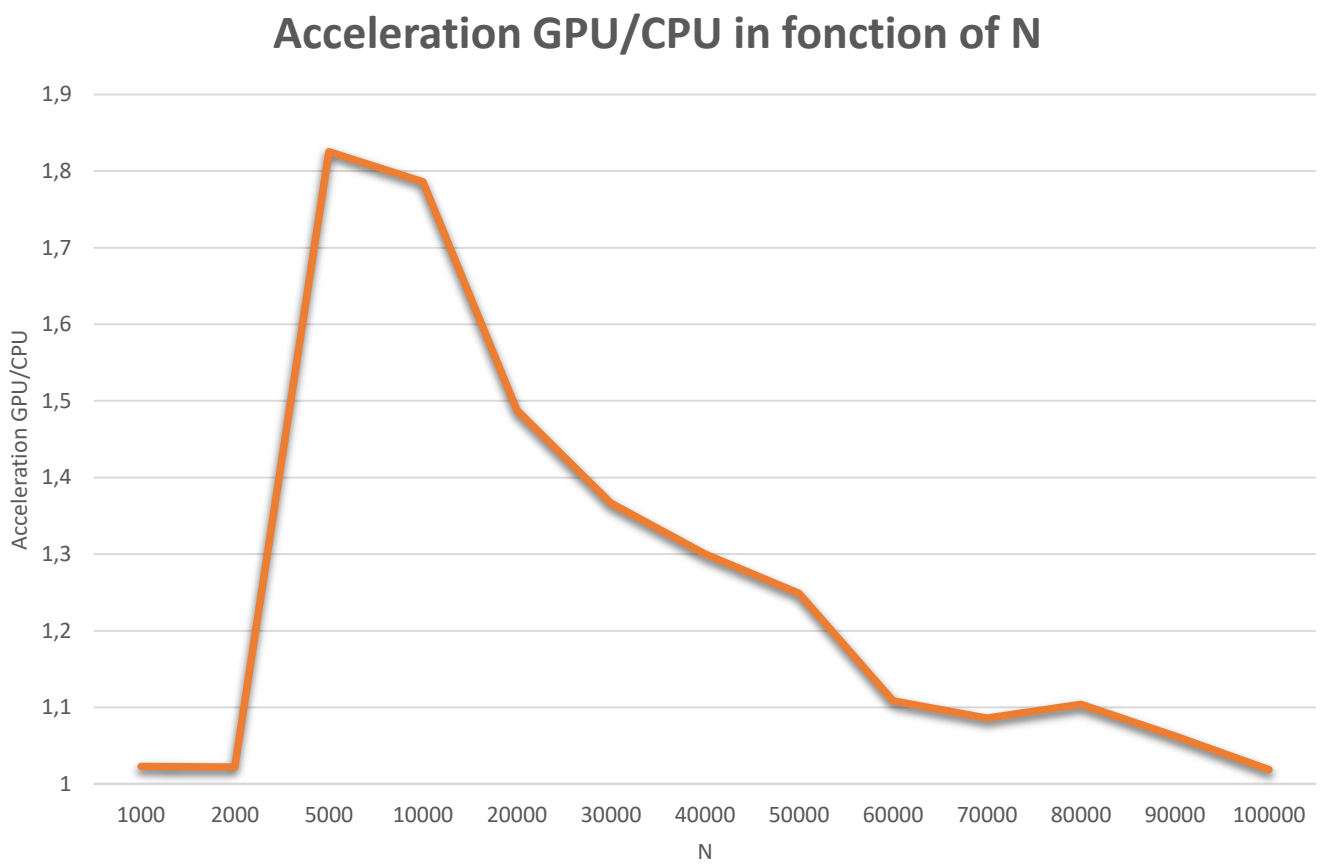


- Test with random array of size N (choice)
- Test with file reading
- Test with a dataset

Used cars dataset

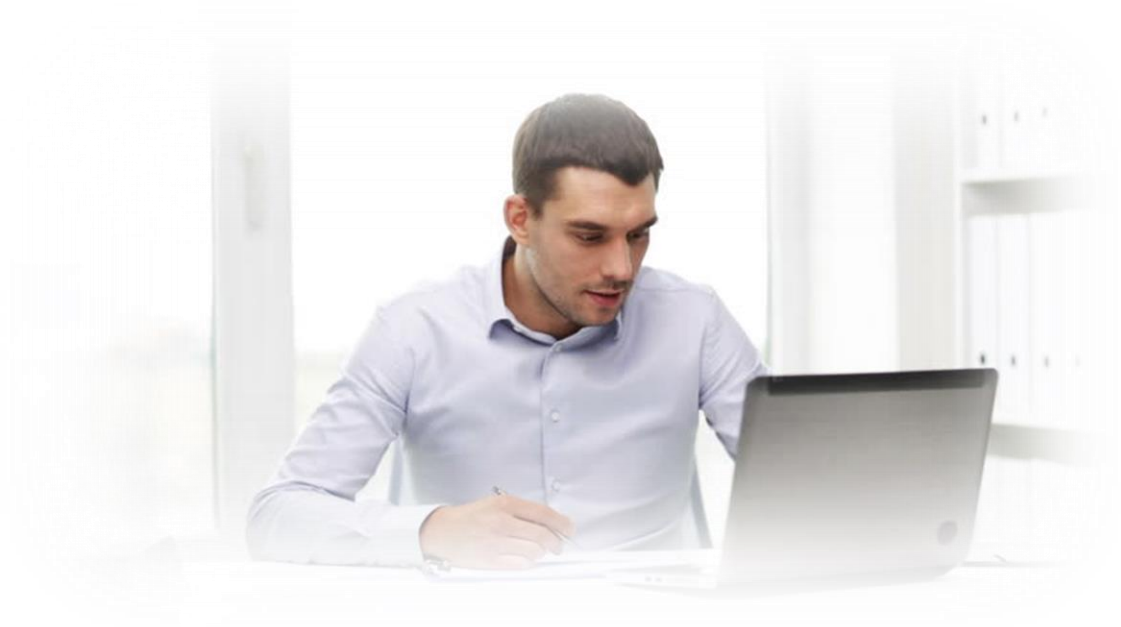
- Data preprocessing with python-pandas
- Data reading
- Algorithm application
- Output:
    - Top 10 most expensive cars
    - Top 10 most kilometers

Performance analysis of our implementation



How can we improve our program

- Parallelization of the sort
- w-wide binary search ?
- w-partition search ?



Thank you for your attention