UNIVERSITÀ DI PISA

Computer engineering

Computer Architecture

Word count parallelisation

Francesco Berti

Iacopo Canetta

Giovanni Marrucci

ANNO ACCADEMICO 2022/2023

Contents

1 Introduction

The following sections analyze the process of creating a word count application for CPU and GPU, that is an application to count the total occurrences of all different words present in a document. The goal was to find an algorithm which is both fast, in terms of execution times, but also scalable through parallelisation so it can be used for bigger documents.

1.1 Sequential procedure

The sequential algorithm consists in taking in input a document, from which all words are read and saved into memory. Then the words are read in sequence and a structure keeps track of the number of occurrences for each of them.

1.2 Parallel procedure

Nowadays with the growth of AI, elaborating text and being able to recognize and extrapolate words that are used more commonly are needed to be as fast and efficient as possible. Improving the efficiency of text processing algorithms through parallelisation can help us training Ais faster and feed them bigger documents.

Given an input document containing the text, it is read and saved sequentially. The count of the occurrences, however, is parallelised: each thread receives a portion of the M words, the number of words received is $\frac{M}{N}$. After every thread has finished, the main thread joins all of them and merges the results, summing all the occurrences.

2 Test Machine

The computer's specifics used for testing the problem and analysing the performances effect the optimisation and the bottlenecks. In order to keep the results coherent with each other we used the same machine for the whole project. Below are shown the specifics of the system used:

- AMD Ryzen 7 5800X 8 cores 16 threads 4.6 GHz Cache L1: 512KB (8-ways) Cache L2: 4MB (8-ways) Cache L3: 32 MB (direct-mapped).
- NVIDIA RTX 3080 10GB GDDR6X 8704 CUDA cores.

3 CPU Solution

The first implementation is designed to only use the computational power of the CPU, trying to get the best possible performance with its limited number of cores. Our analysis used algorithms based on different structures, seeing the pros and cons of each of them and how they influence performance, which are evaluated mostly in terms of time. We also performed some optimizations in order to eliminate the bottlenecks of the best performing algorithm.

3.1 Data sets used

Before showing the solutions proposed, it is necessary to analyse the inputs for the problem since they heavily impact the results. Initially the chosen inputs for the problem were these texts, representing possible cases:

- All words are the same, representing the best case.
- The book *Moby Dick* by *Herman Melville* and a collection of books, representing the average case.
- All words are different, representing the worst case.

Moreover, for the first and third cases we created different files dimensions to see if the speedup would change depending on the length of the file analysed. The file sizes for the actual tests were: 1 *KB*, 10 *KB*, 100 *KB* and 1 *MB*. The best case scenario was useful, together with the worst case scenario, to determine the execution time span for each solution. Nevertheless the best case is not included in this documentation since it's not interesting from the speedup analysis point of view. Consequently, to ease the documentation and be more straightforward only the following tests are evaluated in the documentation:

- All words are different (Worst case): 1 *MB*
- Moby Dick (Average case): 1.2 *MB*
- A collection of books (Average case): 100 *MB*

3.2 Structures used

For the CPU implementation, different structures have been used by the threads to keep track of the words' count. Every implementation effects in its own way on the performance for different reasons. Each thread uses a structure in which the start and end of the interval of words to analyze is reported as well as a structure to return the results. The returning structure used are:

- List: the *List* structure was included to show how the choice of the right structure to elaborate data can effect the performance. In fact, due to the multiple pointer dereferencing to search for a specific word, it was expected to have high execution times and a larger number of misses in comparison to the other structures.
- Vector: this structure has the advantage to dynamically add a new element but it also requires to iterate through all the vector for every new word encountered.
- Map and Optimized map: this structures have been chosen to allow a $O(1)$ access time when searching for a word. The *Map* and *Optimized map* solutions are similar, but the difference is in the merging: the first case uses an iterator to find words, while the second case directly uses the index. For this reason, the second version is expected to have significantly lower times and higher speedups.
- Tree: this approach is designed to reduce merging time, which was found to be the main bottleneck. It is based on a perfect binary tree structure in which the leaves count the occurrences of the words and the parents merge their results. Considering h equal to the height of the tree, this algorithm uses $2^{h+1} - 1$ threads and $2^h - 1$ leaves. In this solution, merging time was expected to be reduced, but in order to ensure the correct results some form of thread synchronization was used (in particular mutexes) and that could impact the overall performance of the algorithm.

3.3 Evaluation of the results

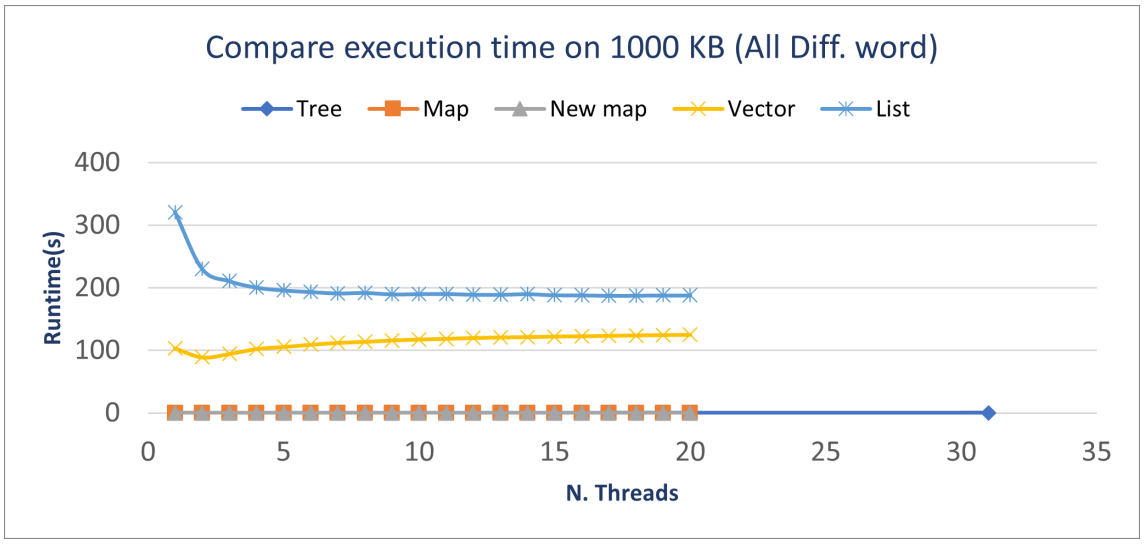
Every algorithm has been tested 20 times for each different number of threads used during the execution. The *Tree* solution has been tested changing the height of the tree from 0 to 5 (which means 1-3-7-15-31 threads), whereas all the others solutions were tested using up to 20 threads. This values are chosen to check how the use of the physical threads of the CPU effects the times and speed up, while using more than the number of physical threads slows down the execution due to the fact that the threads in addition can't be executed in parallel and need to be scheduled.

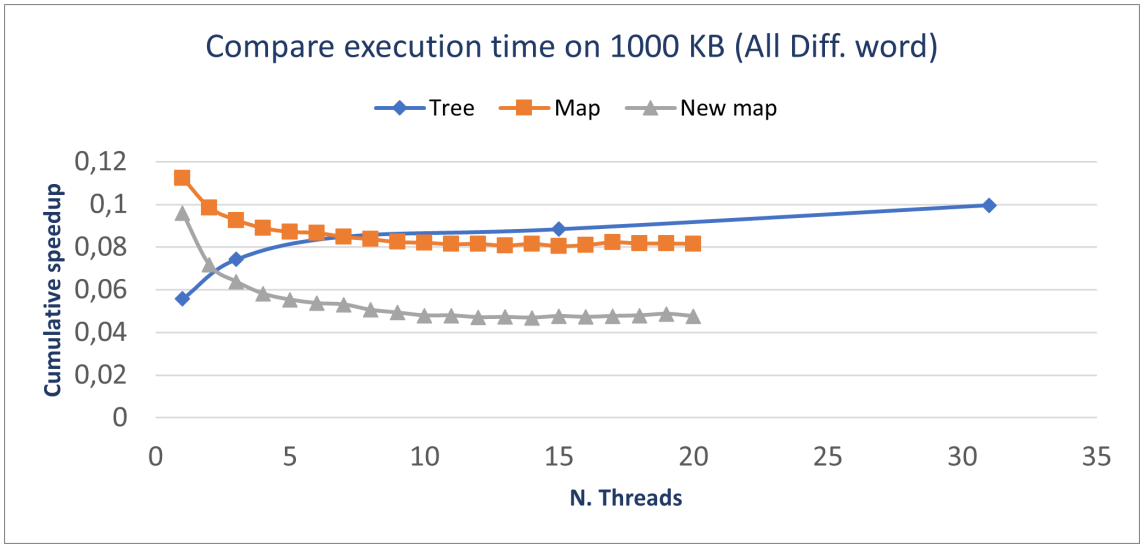
To measure the execution time of each solution the C library function *clock()* was used, it returns the number of clock ticks elapsed since the program was launched.

In the sections below are shown the results obtained using the samples previously mentioned:

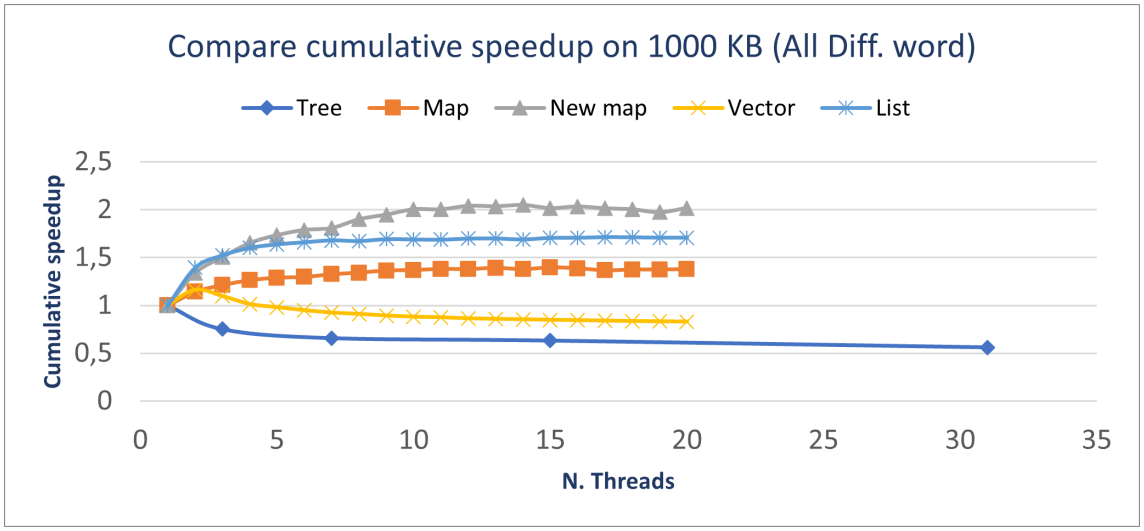
3.3.1 Worst Case

As shown in the graphs below, the fastest algorithms were the ones based on “Map”, “New map”, and “Tree” structures; the second graph shows a zoom of the times of those algorithms, showing how the map solutions have a better scaling.





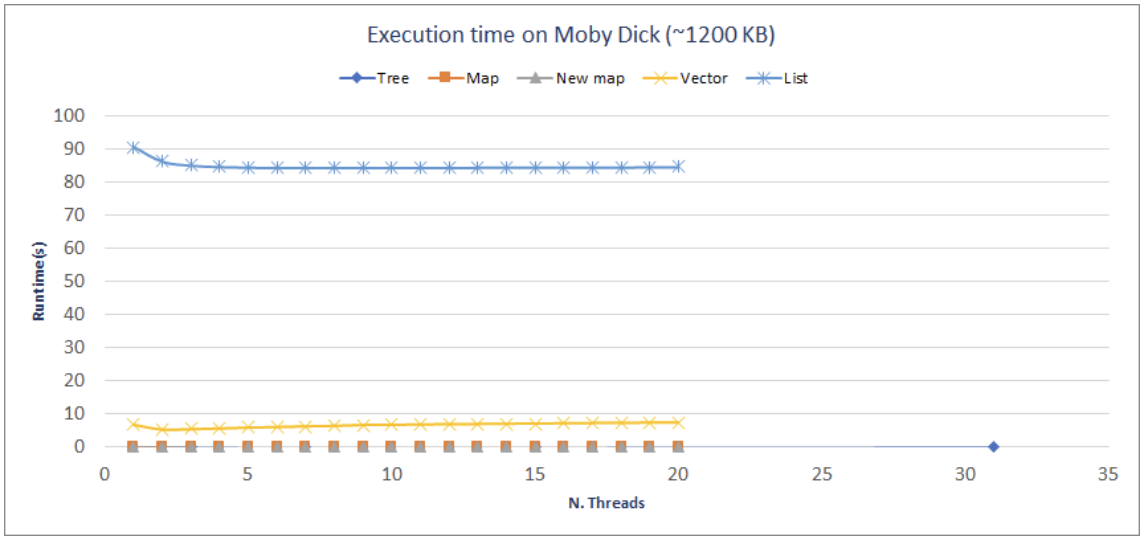
A good way to analyze the structures behaviors is also the cumulative speedup, as shown in the graph below. The best results are achieved by the “New map”, followed by “List” and “Map”.

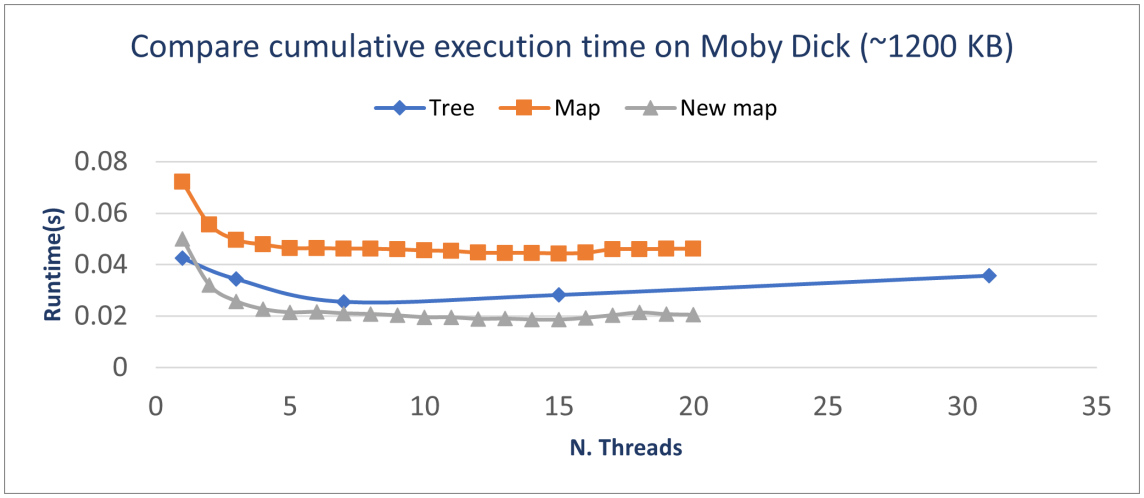


These results are not enough to justify the use of any of these parallel algorithms as supposed to the sequential one: the speedups are not high enough. However this is the worst case scenario in which all the words are different, clearly not a real scenario. It's clear from these graphs that the *List* and *Vector* solutions perform poorly compared to all the other solutions.

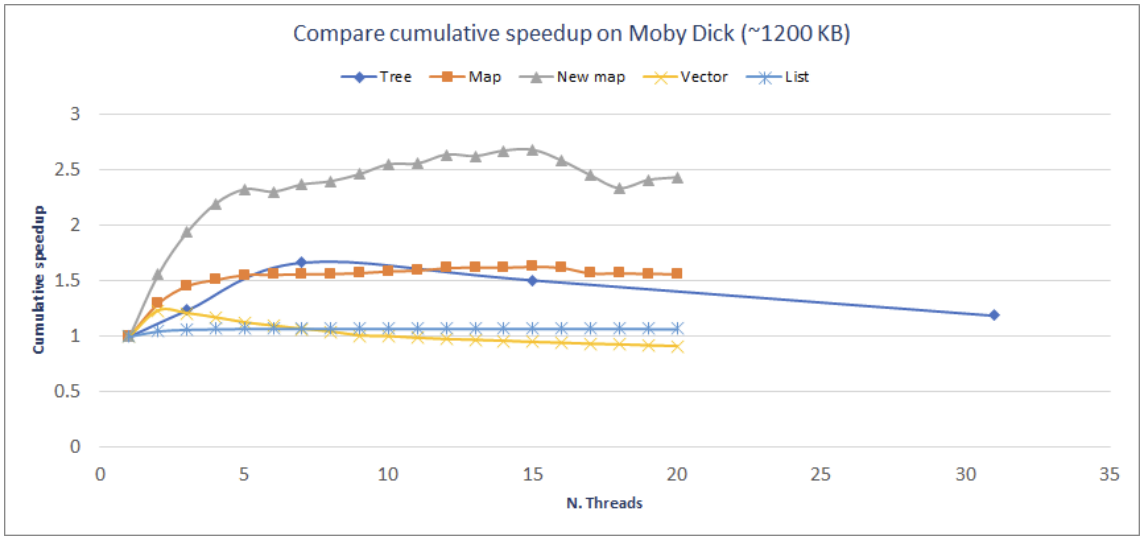
3.3.2 Average Case

In the first graph shown below it can be observed how every solution, except the *List* one, is able to guarantee a reasonable execution time



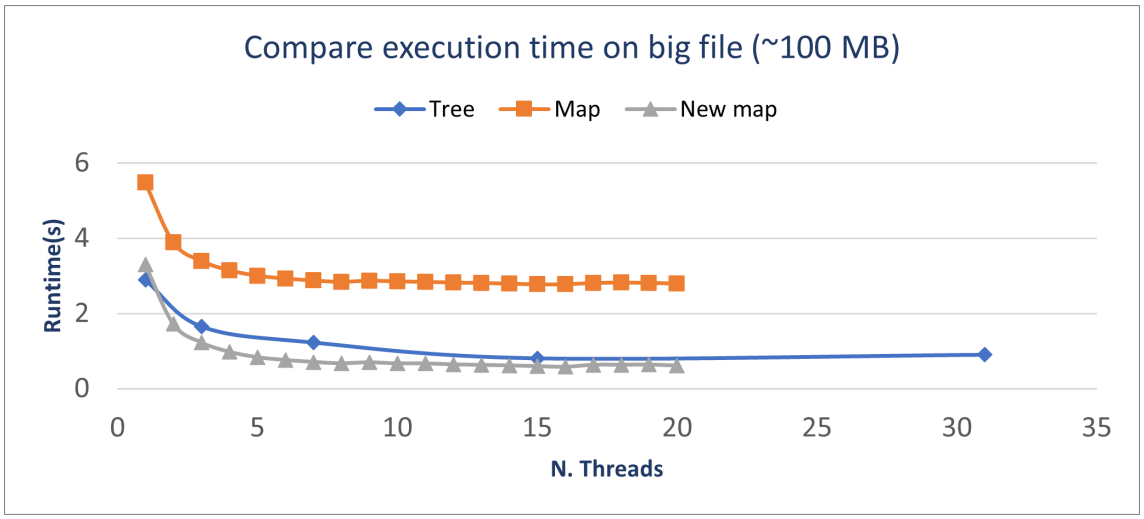


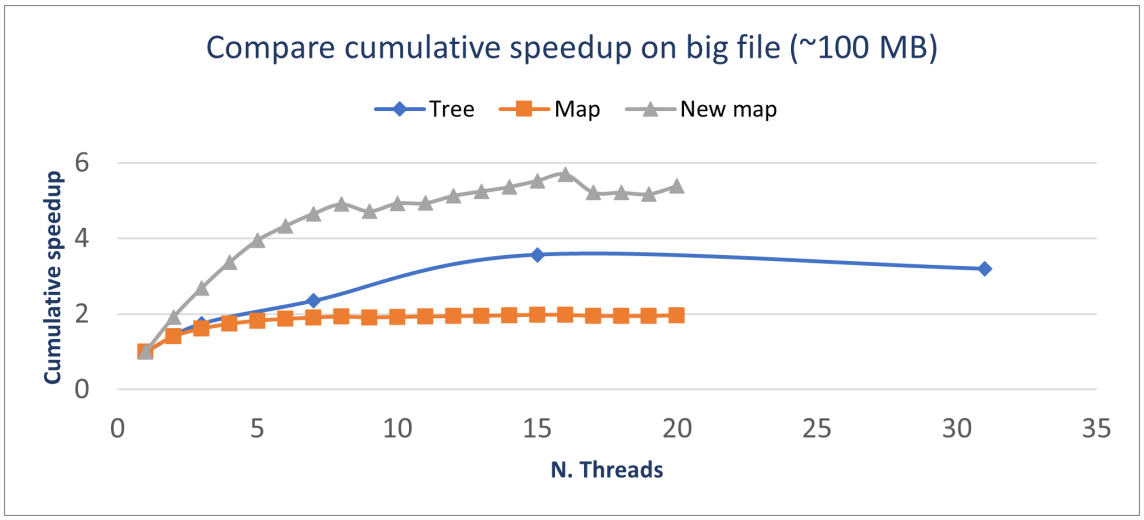
Taking a look at the cumulative speedup in the graph below, the best result is achieved again by the *New map*, followed by the *Map* and *Tree*.



3.3.3 Collection of books

After having seen the results in the average case it was decided to use a larger file, a collection of books, with possibly more repetition. The *New map* was again found to be the better perform both in terms of execution time and cumulative speedup. It can be observed that the *Tree* solution scales better than the normal map and has execution times comparable to the *New map*.





3.3.4 Best algorithm for CPU

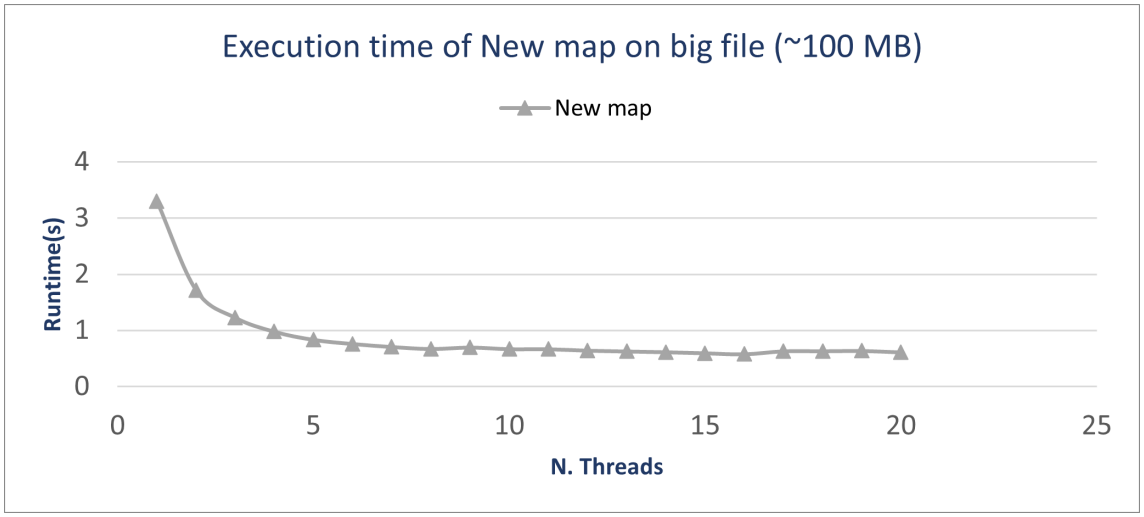
After these analyses, it is clear that:

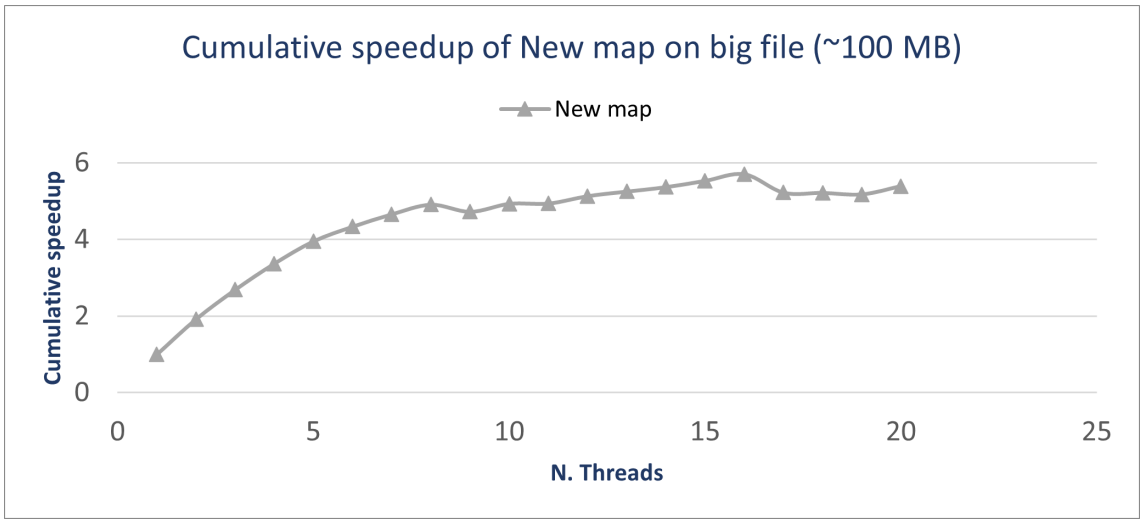
- *List* has too large execution times, this is due to all the dereferencing done to interact with the list structure. The cache misses were measured using Valgrind's tool "Cachegrind" and found out to be 2.3% whereas in all the other solutions the miss percentage was $< 1\%$, confirming the expectations.
- *Vector* doesn't scale and this is caused by merge operation in which all the returning vector of each thread have to be parsed, so in the worst case scenario this procedure has a O(n) time complexity.
- The *Map* and *Tree* both have low runtimes, but the speedup is not high enough, especially for the *Tree* solution in which the cumulative speedup reaches values smaller than 1 for both the average and worst case.

With these considerations we decided to sweep aside the "Vector" and the "List" for out last examination since they had not good execution times and speedups.

3.3.5 Closer Look to New Map

The graphs below show a better look for the *New Map* behaviors. From this analysis it is possible to get the lowest execution time, equal to 0.5778s, and the max cumulative speedup, equal to 5.71, both reached using 16 software threads. This result was expected given that the number of physical threads of the CPU is equal to 16.





3.4 CPU Conclusions

In conclusion, we can say that the best performing solution was the *New Map*. As expected, the use of a map structure offers a big advantage when it comes to merging the results from the threads thanks to the O(1) access times, while all the other solutions require to search an element every time, which has a O(n) time complexity. Moreover, some other interesting results were reached, since the *Tree*'s use of the mutexes had worsen the results due to the overhead for the communications.

4 GPU solution

The word count application is difficult to realize in the GPU realm since there is not even a string object nor utility functions like *strcmp()*. For this reason, all the solutions proposed in this section count *integers* instead of *words*; the problem is basically the same since, in order to use these solutions to count words, it would be sufficient to find an hash function that converts every word into an integer.

Five different solutions are proposed, the principle is the same applied to the CPU: divide the data set into smaller chunks which are analyzed by different threads.

4.1 Data sets used

Since the application counts integers instead of words, the datasets used to perform benchmarks need to be changed. The worst case scenario is not included since it is not a realistic case and the max execution times were already been established in the CPU side; for this reason the datasets analyzed all represent average cases:

- *Small data set*: a total of 1M elements with 32K distinct elements, a small data set to compare all the version together
- *CPU comparable data set*: a total of 18M elements with 126K distinct elements (0.67%), this data set is comparable to the collection of books used in the CPU analysis which was the most interesting case
- *Large data set*: a total of 500M elements with 50M distinct elements (10%), used to show the potential of the best solution

Each data set, except *CPU comparable data set*, was built using CUDA functions to generate uniformly distributed random values. The second data set was instead built by mapping to each different word found in the collection of books an integer.

4.2 Solutions

Since one of the most limiting factors in this case is the amount of data to transfer, every solution aims to reduce that. In particular, every solution returns the results using one big transfer instead of multiple small ones. In every solution, each thread analyzes its interval and returns the result in a structure. Five approaches were used:

- Version 1: each thread has its own array of *MAX_ELEM* elements (where *MAX_ELEM* is the biggest number in the data set), and the CPU will perform the final merge using an unordered map. Clearly this approach cannot be used for large files: the amount of data to transfer scales linearly with the number of threads and *MAX_ELEM*.
- Version I: an array of struct (number, count), whose length is the same as the amount of integers in the dataset, is passed to the threads. This solution however uses multiple *if* statements. Since the GPU is composed by single issue cores, this sections will be executed sequentially, leading to expected execution times higher if the compared to other solutions.
- Version 2: it works in the same way as the first version, but the first thread of each warp performs an initial merge of the results. This operation should decrease significantly the amount of data to move and to merge at the CPU, leading to better performance.
- Version 3: an array of integers, whose length is equal to the amount of integers in the dataset, is passed to the threads. Each thread uses *atomicAdd()* to increase the count while ensuring consistency. This solution uses way less memory than the others and has the benefit of eliminating completely the merge.
- Version 4: it works exactly like the previous version but the array's length is *MAX_ELEM* so the amount of data transferred is reduced even more.

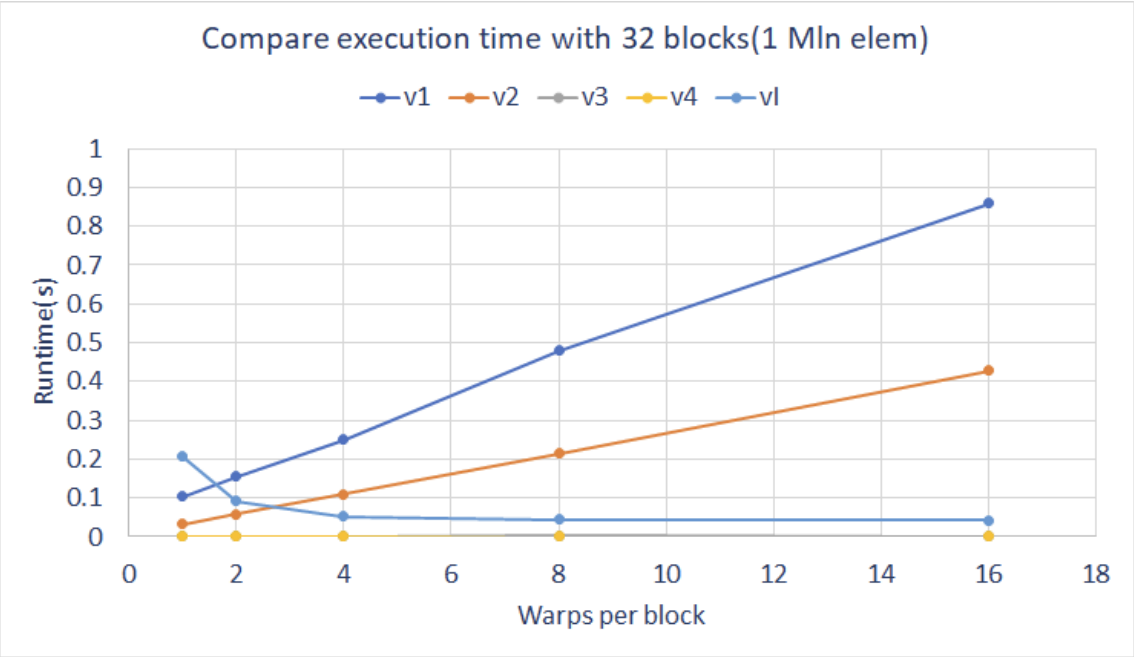
4.3 Evaluation of the results

Every solution has been tested 20 times for each different block sizes and numbers of blocks. The basic unit used is a warp, 32 threads. The first four solutions were tested by doubling every time the amount of warps used, whereas for the *version 4* every multiple of the warp size was used to better represent the performance.

To measure the execution times the C library function *clock()* was used just before launching the kernel and after having all the data ready to use (so it also includes the time to copy data).

4.3.1 Small dataset

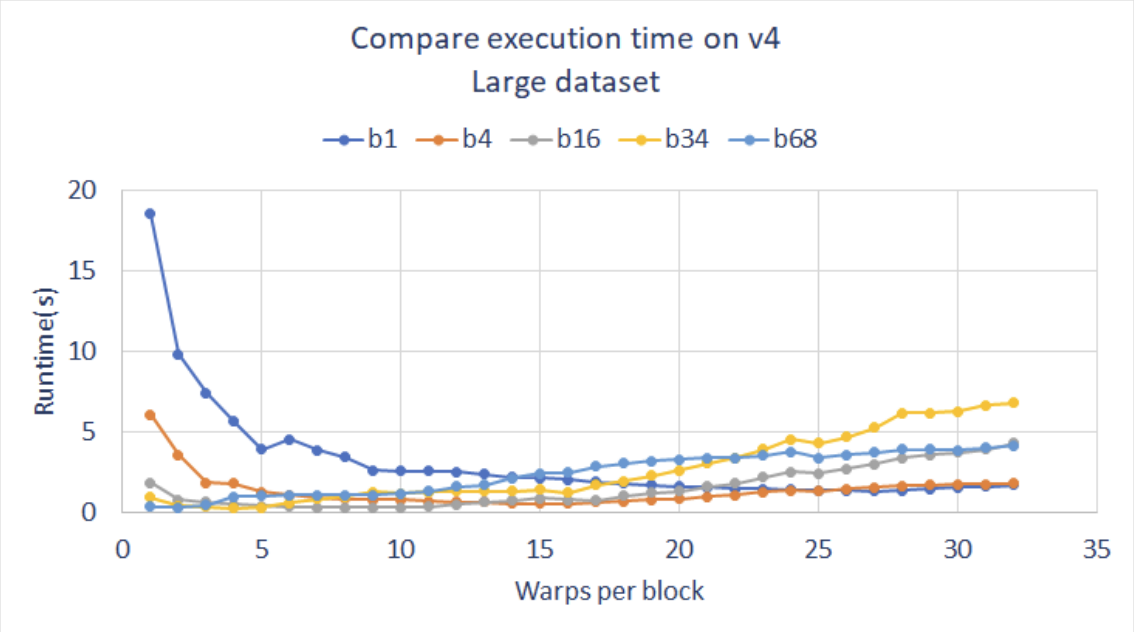
The graph below shows the execution times for each different solution on the *small data set* using 32 blocks. As explained in the previous section, *version 1* and *version 2* both scale linearly with the number of threads, even if for *version 2* has the angular coefficient smaller due to the initial merging inside the warps. This is the reason why 32 blocks were chosen: it corresponds to the maximum amount of blocks that can be used on this data set before *version 1* and *version 2* fail to run due to insufficient memory. On the other hand, *version I* scales linearly with the dataset's size, so an higher number of threads means that each thread is going to handle a smaller amount of data and the execution time improves. Clearly *version 3* and *version 4* are the best performers in this case, since they are pretty much the same algorithm (the only difference is that *version 4* uses less data). The next graphs will focus solely on *version 4*.



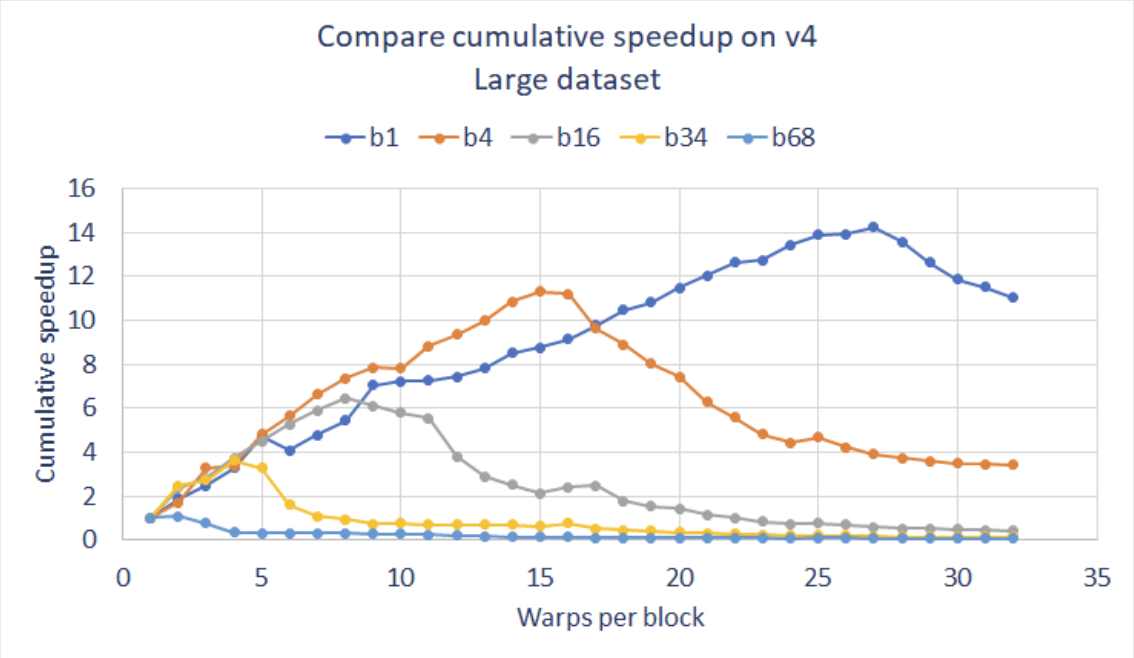
4.3.2 Large dataset

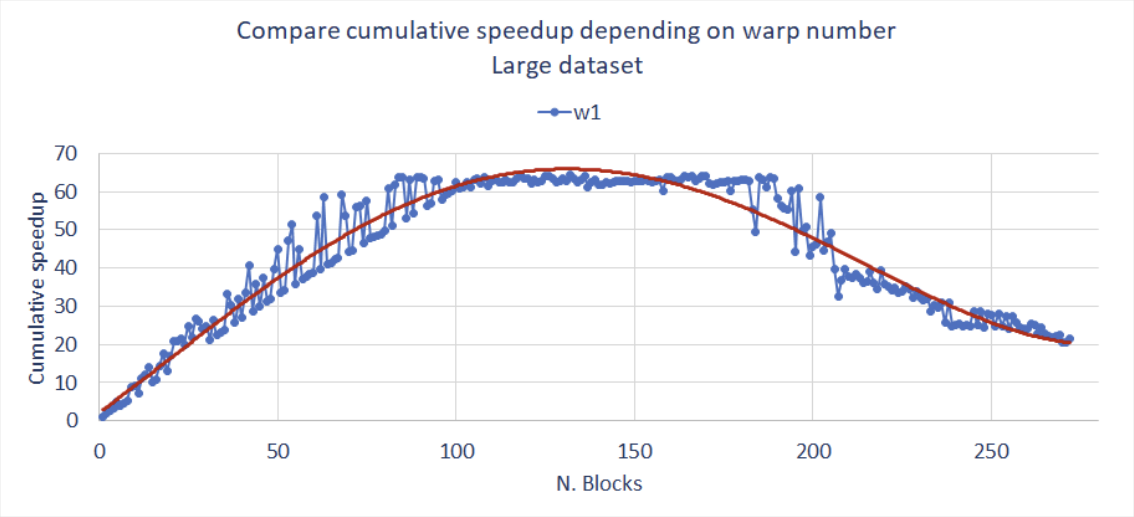
The last three graphs show the potential of the GPU solution using the *Large dataset*. The execution times graph highlights how, with a higher number of blocks, the number of total threads increases so quickly that it surpasses soon the number of physical threads. This, combined with a higher degree of synchronization needed to handle the higher number of threads, has as consequence that the execution times start to increase. The best performance is achieved using 34 blocks and four warps for a total of 4352 threads with an execution time of 0.25s.

Given this result and the ones on the *CPU comparable data set* is reasonable to expect that, if the dataset was even larger, the point of minimum would be reached for a higher number of threads.



The next two graphs underline how the speedup is effected by scaling the block size (or warp number) and the number of blocks. In the first graph the maximum cumulative speedup of 14.24 is achieved for one block and 27 warps (864 threads in total), whereas in the second graph the maximum of 64.35 is in 118 blocks with one warp. The reason why the achieved cumulative speedup is better with a higher number of blocks is caused by the number of CUDA cores present in each SM - Streaming Multiprocessor, which is 128 on our hardware. For instance, by using 68 blocks with one warp the total number of threads is 2176 and each SM will be able to execute 32 threads in parallel. The same amount of threads can be reached by using 4 blocks and 17 warps, but in that case the number of threads allocated to each SM would be 544, so most of them cannot be executed in parallel but need to be scheduled. So if the choice was between increasing the number of threads per block or the number of blocks, the second option is by far the best, at least in terms of execution times.

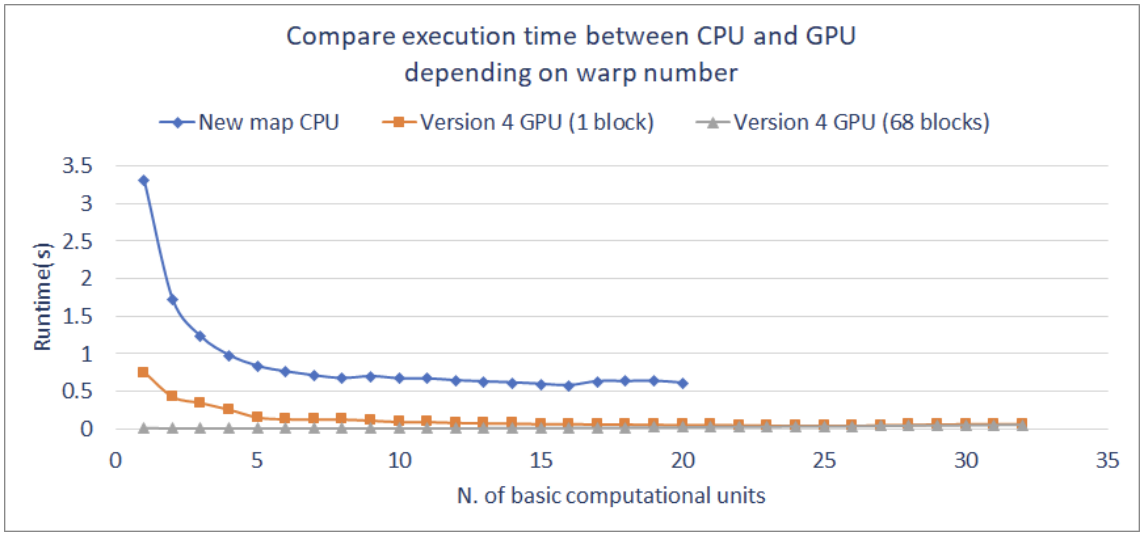


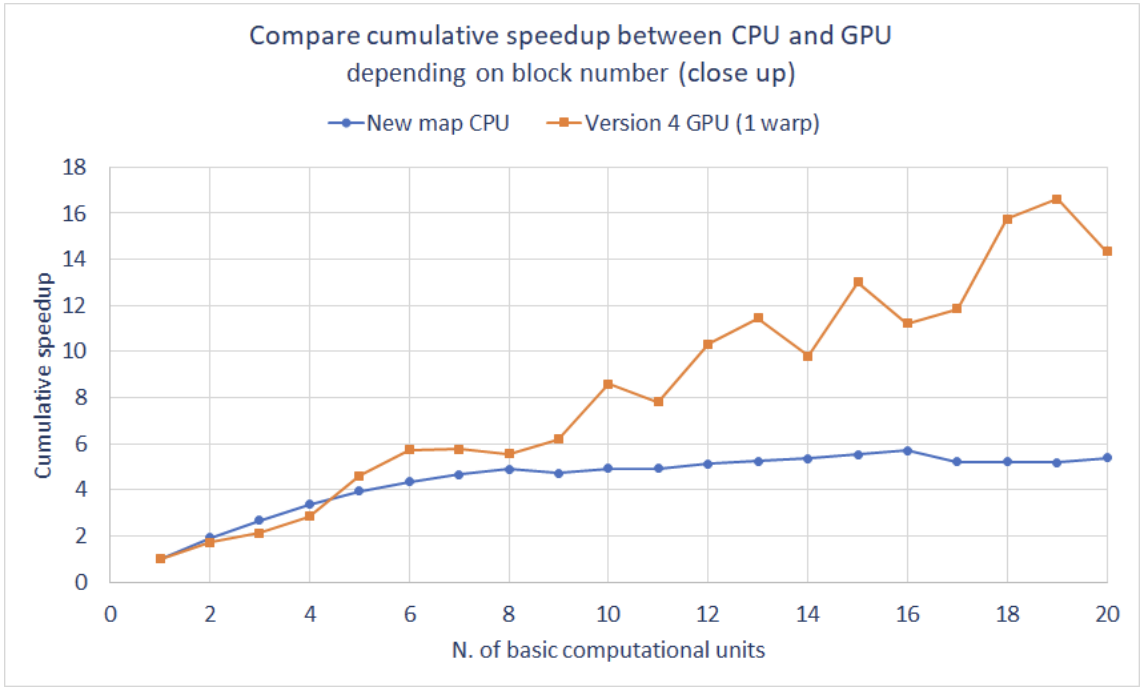


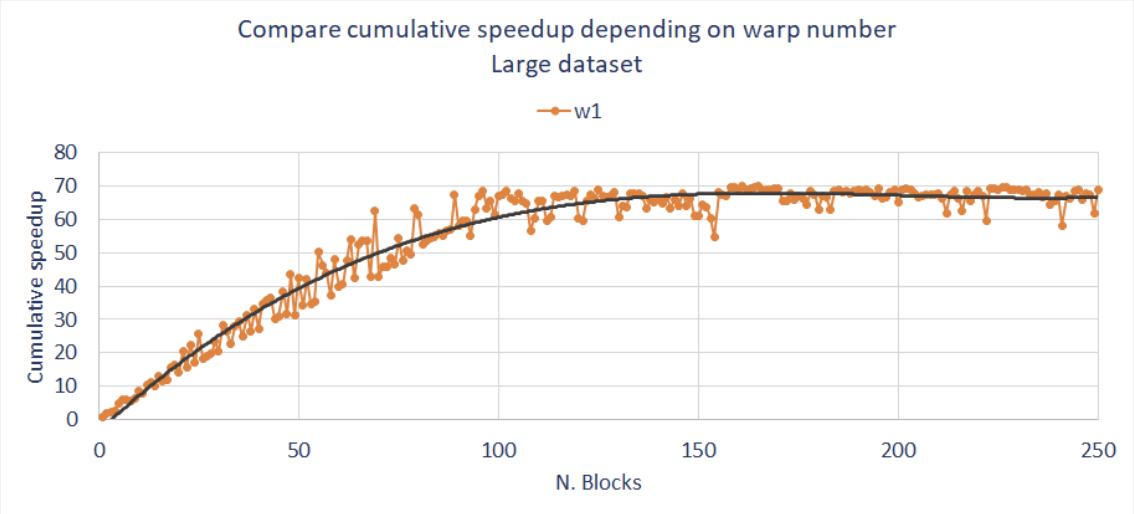
4.3.3 CPU v/s GPU

The following graphs compare the performance of the CPU and GPU's best solutions over the *CPU comparable dataset*. The first graph shows how the execution times are reduced from the beginning using the GPU solution. The GPU solution also offers better scalability as shown in the second graph. In fact, when using 16 blocks and one warp, the achieved cumulative speedup is 11.57 whereas in the CPU it was 5.71. The second graph also includes the worst cumulative speedup, which was achieved using 32 warps, in that case the amount of threads is too high for the analyzed dataset so there's no improvement in terms of speedup since the execution times are already low from the beginning.

To show the complete performance of the GPU on this dataset the third graph was included, the maximum cumulative speedup of 69.89 was reached for one warp and 161 blocks.


Compare execution time between CPU and GPU
depending on warp number
New map CPU
Version 4 GPU (1 block)
Version 4 GPU (68 blocks)
Runtime(s)
3.5
3
2.5
2
1.5
1
0.5
0
0
5
10
15
20
25
30
35
N. of basic computational units


Compare cumulative speedup between CPU and GPU
depending on block number (close up)
New map CPU
Version 4 GPU (1 warp)
Cumulative speedup
18
16
14
12
10
8
6
4
2
0
0
2
4
6
8
10
12
14
16
18
20
N. of basic computational units



5 Conclusions

The word count application is best suited for CPU since they're more general purpose and implement various methods to handle strings, however their small number of physical threads could be a limiting factor when analyzing big files (e.g. 1+ GB), in that case it makes more sense to use a GPU. In both CPU and GPU the parallel approach is better than the sequential one with a max cumulative speedup of 5.71 for the CPU and 16.16 for the GPU in the collection of books scenario.