Universitat Politècnica de Catalunya

Facultat d'Informàtica de Barcelona (FIB)

TGA project

Mergesort

Author:
Guillem Pérez Oltra
Pere Vergés Boncompte

June 20, 2020

Contents

1 Introduction

In this project, we have decided to implement with CUDA the sorting algorithm, mergesort. You are probably already familiar with this basic algorithm, but we are going to make an explanation of how this algorithm works and how is usually implemented.

1.1 Mergesort

Mergesort is one of the most common sorting algorithms, now I will state the most important characteristics of this algorithm:

- **Divide and compare algorithm**, it is a divide and conquer algorithm which means that it is based on multiple branch recursion, splitting the problem into smaller sub-problems. In the case of the mergesort the strategy followed is to split the unsorted list into smaller ones, when we reach a leaf or we make a threshold then we sort the list and do start ordering all the sub-problems until all we have done so for the whole list, this property makes this sorting algorithm very easy to parallelize.
- **Comparison based sorting algorithm**, which means that the algorithm sorts by comparing the elements.
- **Stable**, most of the implementations of the mergesort create a stable sorting, these means that if there are two or more elements that have the same value, they will be ordered in the same way the input was introduced.
- **Average performance**, As most of the sorting algorithms this has an average performance of:

  $$O(nlogn)$$

  In the worst-case scenario, the performance will be the same as the average. The best-case scenario would be the list already ordered in which case the performance would be:

  $$O(n)$$

- **Space** The space this algorithm uses in all the cases is:

  $$O(n)$$

- **Jhon von Neumann**, invented this algorithm in 1945.

2 Algorithm

Now I am going to explain how the basic mergesort algorithm works. First of all, I will show the pseudo-code of the algorithm and then I will explain how does it work.

Algorithm 1: mergesort(arr,l,r) pseudo-code

```
if r > l then
    Find the middle point to divide the array into two halves;
        middle m = (l+r)/2 ;
    Call mergesort for first half;
        mergesort(arr, l, m);
    Call mergesort for second half;
        mergesort(arr, m+1, r);
    Merge the two halves sorted in the previous steps;
        merge(arr, l, m, r);
end
```

Since this is a divide and conquer algorithm the pseudo-code is showing a recursive implementation. First of all, I will explain the parameters:

- **arr**: Is the array that we want to sort.
- **l**: It is the position of the array where we start to sort.
- **r**: It is the position of the array where we stop to sort.

The implementation checks that the start position is smaller than the end position to sort $r > l$, and that the size of the sorting is larger than one since it is one the array is already sorted.

If this condition is met then we first find the middle of the array to start with the recursive calls.

Once we have the middle point we first call the recursion for the first half of the vector which is the left part of the array, so as arguments we are going to forward the array, the start index, and the finished index which is going to be the middle point.

Then we do the same for the second half of the array, which is the right part, in this call we are going to forward the array, the starting point which is the middle plus one position and the end.

Once the recursion is met we are going to receive the two halves ordered and

then use the merge function to merge the sorted arrays we have received and return the array sorted.

The following figure is a very good and intuitive representation of how the merge sort works:

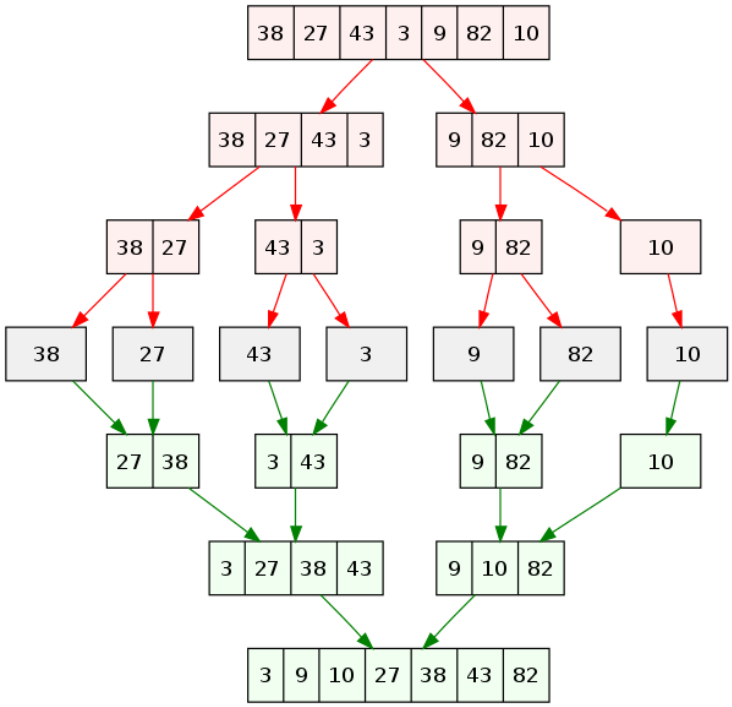

Figure 1: Mergesort

Performance We took the sequential code from kshitiz38 github merge_sort_cuda repository. We have run the sequential algorithm several times so we could compute the mean of all executions. Some things we need to have in consideration. First, the algorithm was executed in the Boada server, so when we have the cuda version implemented we can compare running times with as much precision as possible. Second, we used an array with 4,145,152 random elements between 0 and 4,145,000. With that said, the mean of our running times algorithm is 0.621066 seconds.

Possible improvements This is the most basic and simple but there are easy improvements that can be done:

- **Threshold** We can add a threshold to stop the recursivity do not reach the leaves. Once we reach this threshold we apply quicksort and sort the small chunk of the array. This is going to be an improvement since we are no going to do too much recursion in the case we want to parallelize the code we will reduce the number of threads needed, which means that we are going to reduce the dependencies in the code.
- **Parallel implementation** Another way of improving the implementation of mergesort is doing the computation concurrently, we are going to take this approach. One of the ways of doing such task is to have multiple threads doing the calculation, have in mind that when we are going to have dependencies when splitting the array, and when we reach the leaf or threshold we are going to have as well dependencies when merging the split arrays, so when approaching the parallelism we have to optimize the way tasks are assigned to threads to waste the minimum amount of time between dependencies.

3 CUDA Implementations

In this section, we are going to describe all the versions of our solution to the mergesort problem implemented by CUDA. So first we will do a basic implementation of the problem and once we have the basic up and running we are going to enhance it adding improvements to it.

3.1 First version (*mergesortV1.cu*)

The mergesort algorithm is divided into two main functions, the first function is the one that splits the array to be sorted in smaller arrays and then we have the merge function that merges the two ordered arrays.

So we have split the problem in the same way. First, we have the splitting function, to make this part parallel what we have done is to divide the array into chunks, and each thread is going to apply mergesort to its assigned chunk. This way we have that each thread applies the mergesort algorithm to a portion of the array sequentially. After finishing this first part what we are going to obtain is the input array ordered by chunks (each chunk is going to be ordered, but not the whole array) so it will be semi-ordered.

To achieve that, we run into a few problems. The first thing we had to do was to check if recursion is allowed in cuda, because we did not take a look at lab lessons and our sequential algorithm uses it. Cuda methods and functions need to be declared as global, host, or device. It means you have to specify where your cuda function is going to be called and where it is going to be computed. Global means the function is going to be computed in the device but called from the host, device means called from and performed in device and host means called from and performed in the host.

So, the recursion problem is that first is going to be called from the host and the other ones will come from its function. To solve this, we split the function into two. The first one named "callMergeSort" is the one called from the host. Each thread will first run this code before the merging process. In this function, which is declared as global, we obtain the thread id and assign to the thread the chunk of the array it has to be ordered, and then each thread runs the "mergeSortKernel" which will compute the sorting and merging process.

For the second part of the problem what we have to do is to make all these chunks merge into a whole ordered array. Since this is the first version we decided to do this process sequentially, what we do is start ordering the first two chunks and then merging what we already have ordered plus the following chunk.

With this approach, the mergesort algorithm works perfectly fine.

Performance As we did with the sequential algorithm, we have run the algorithm a few times in Boada server, with an array size of 4,145,152 including random numbers between 0 and 4,145,000. We used 16 blocks and 256 threads per block for now. The mean time for the GPU procedures we obtained is about 91.4652 milliseconds. We have to take into consideration that approximately 80.98% of that time is the time used to transfer the array from host to device (6.27%) and from device to host (12.75%). Furthermore, the mean of the time for computing the merge algorithm in CPU is about 1.074 seconds. So, the total running time is 1.165 seconds. As we can see, the cuda version is not faster than the sequential one yet. But, the main goal of using CUDA and GPU instead of only using CPU, is making the most of parallelism so we can hide the enormous bandwidth when transferring data from host to device. We will try that in the versions below.

3.2 Most suitable number of blocks and Pinned Memory (*mergesortV2.cu*)

If we want the parallelism to fully exploit, we first need to find a proper number of bocks and threads per block. The maximum number of blocks we can use is 65535 per dimension, which is 65535x65535x65535 blocks. In this version, we will only use the first dimension. The maximum number of threads per block is about 1024. In this version, we found that number as you can see below in the performance section.

Then we tried to improve the time wasted transferring memory from host to device and vice versa. To reach that goal, we took profit of cuda's pinned memory. With that memory, an important improvement is realized.

Performance We proceed to compute our cuda algorithm several times changing the number of blocks and threads as you can see in the graph below. In the y-axis, we have located the number of threads/blocks. In red, you can see the number of threads per block, in blue the number of blocks. The time in milliseconds is in the x-axis.

But notice we did not include the time spend in the CPU since this time is going to be the same when modifying GPU parameters. So the time we are comparing is the time spent transferring data from CPU to the device, computing the kernel, and then transferring the data from device to CPU. We got the best time with 32 blocks and 128 threads per block, which is 86.585 milliseconds. 24.14% of this time is wasted transferring data.

So then, when we apply cuda's Pinned Memory, the time transferring data from host to device and vice versa is considerably improved. The time trans-

ferring data in both cases after applying cuda's pinned memory is about 1.589 milliseconds, which is the 2.31% (4.62% both) of the total GPU time. The total GPU time is about 68.857 milliseconds. To sum up, our second version is 1.26 times faster than our first one. Not a huge but important improvement.

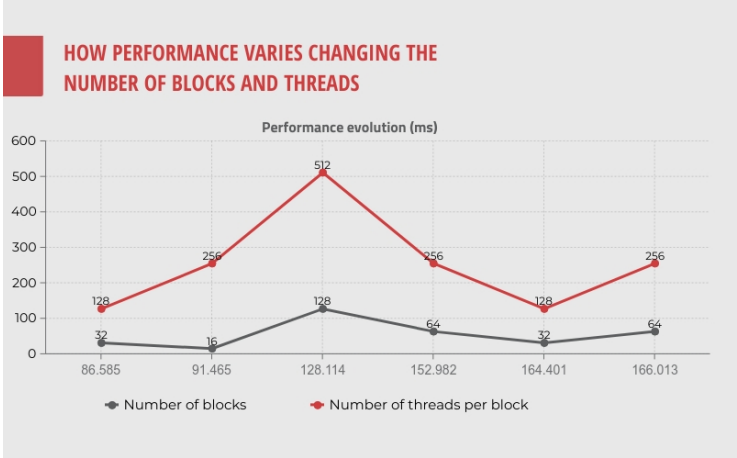

Figure 2: Performance

3.3 Using a second kernel for last part of recursion (*mergesortV3.cu*)

In this third version of the algorithm, we start from the second version, so in this version, we have the improvement of implementing the pinned memory.

The idea in this third version is to implement a kernel to do the last part of the recursion of the merging of the vector. The first kernel we have does the sorting of chunks of the vector and we end up with a semi-sorted vector.
In the previous versions, we took that vector and merged the chunk sequentially and without using recursion.

This new kernel using recursion takes a pair of the chunks that we had ordered and assigned to a thread. Then the next iteration of the recursion will take two of the newly merged chunks (which are double in size of the previous iteration), ordered sequentially, and merged them, the recursion will stop when we reach the point where the chunk is the same size as the whole vector. So bear in mind that for each iteration of the recursion we will double the size of the chunks and divide by two the number of threads.

This will improve a lot the performance of the algorithm since we parallelize half of the algorithm, so we think that this was an important improvement to

make the algorithm perform way better.

Performance We first need to have into consideration that the time spent merging the semi-sorted vector in CPU in the previous versions is about 1.074 seconds, which is 92% of the total time. That means, that improvement here was really needed. The total program time we got after applying recursion and creating the second kernel is about 416.125 milliseconds. 347.2ms (83.44%) is the time spent in merging the semi-sorted vector, 65.745ms (15.8%) is the one spent in semi-sorting the vector and 1.59x2ms in both transfers (0.38% and 0.38%) is the last part. We got an important improvement from the second version; this third version is 2.73 times faster than the second one.

4 Integration with KNN algorithm

To see how our program performs integrated to another problem, we are going to let our code to our classmates Pau Ballber and Laia. The results are shown in a graphic below:

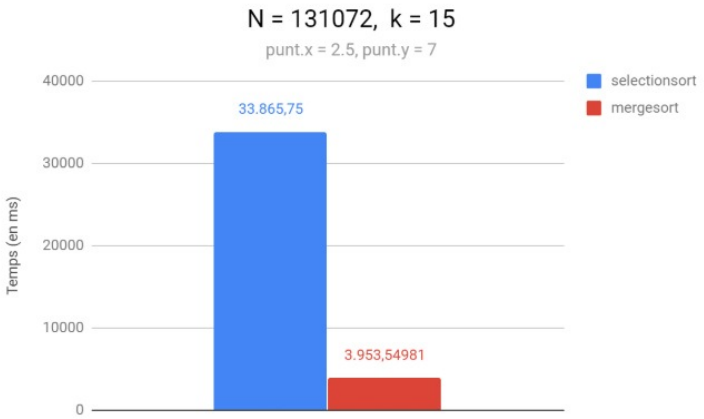

Figure 3: Performance in KNN algorithm.

As you can see, it is a comparision between the sequential selection sort and our algorithm, which performs much better.

5 Conclusions

In this practice, we set ourselves the target of implementing a CUDA's version of mergesort and improve the performance of it. To see in a better way the performance evolution, we put the total running time of all our versions and its speed up compared with the sequential one:

- **Sequential Version** 0.621 seconds
- **First CUDA Version** 1.165 seconds, speed up: 0.533
- **Second CUDA Version** 1.142 seconds, speed up: 0.544
- **Third CUDA Version** 0.416 seconds, speed up (total): 1.493

We can appreciate how the first version in CUDA performs quite worse than the sequential one, this is because half of the algorithm we were applying in the first version was quite worse, this is the second part of the algorithm were we merge the semi-ordered vector. But this first version was to see if we could manage to make the algorithm work using CUDA.

In the second CUDA version, we see a slight improvement in the execution time, this improvement is due to the usage of pinned memory. It is still performing worse than the sequential one because we are using the same algorithm that we used in our first version.

In the last version, we see a huge improvement, this is because we fixed the second part of the algorithm, making it recursive and taking advantage of multithreading with CUDA. In this version, we have an algorithm that performs 1.5 times faster than the original version. I think this is a very good accomplishment and that we have reached the goal we had in mind.

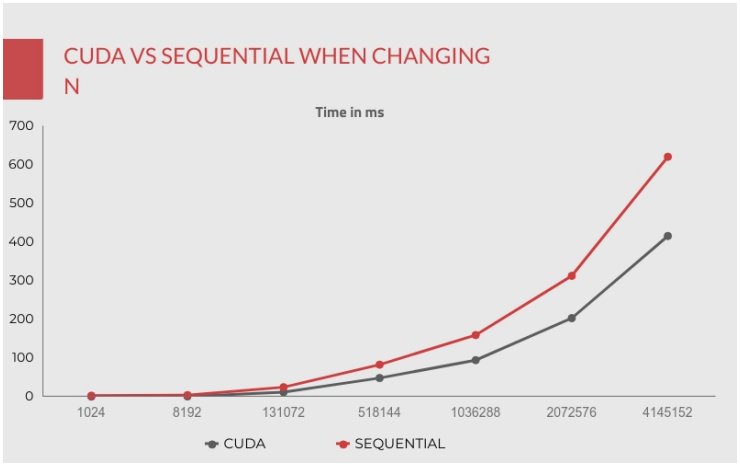

Figure 4: DadesN

In this last figure, we can easily appreciate the improvement of our version in comparison to the sequential version. If we use it for small vectors we do not see this improvement, but when it comes to huge numbers there is a clear improvement.

References

[1] Mergesort Wikipedia [Online]. [22 of April 2020]:
https://en.wikipedia.org/wiki/Merge_sort#Pseudocode

[2] Kshitiz38 Github Mergesort [Online]. [22 of April 2020]:
https://github.com/kshitiz38/merge_sort_cuda

[3] Mergesort geeksforgeeks [Online]. [22 of April 2020]:
https://www.geeksforgeeks.org/merge-sort/