PARALLEL PROGRAMMING PROJECT

A project report of Architectures and Platforms for Artificial Intelligence exam

Master in

Artificial Intelligence

By

GIAN MARCO BARONCINI

University of Bologna

INTRODUCTION

I decided to study two sorting algorithms well known in the literature: Bubble-sort and Merge-sort. In the first case I developed several algorithms to see the differences, while in the second case I directly wrote down the solution that I thought was the best.
I will start by introducing the basic algorithms and the variant I was inspired by to create the parallel algorithm and then move on to the parallel implementation by describing the technical choices.
In all experiments, I used the same seed for the randomisation function in order to make the comparisons between algorithms reproducible and to provide the same complexity to the several sort.
For the array size 50000 was chosen as a fixed dimension, because I imagined that such a system would be used for large computations and it did not make much sense to test it on small arrays, where, of course, there would be no gain in performance.
The assignment of global device constructs and cpu functions were carefully chosen to improve computation.
Finally, the algorithms were evaluated by calculating an indicative speed up between the basic sequential solution and the parallel solution concerned. Execution times were calculated from the moment before the start of an algorithm to the end of the algorithm, as seen in the lecture.

CHAPTER 1
BUBBLE SORT

1.1 The algorithm

Bubble sort is a simple algorithm for sorting lists of data. In it, the data set is scanned, each pair of adjacent elements is compared and the two elements are reversed in position if they are in the wrong order. The programme goes on until the array is exhausted, considering one less element at each step, being sure to take the largest one to the last place at each cycle.

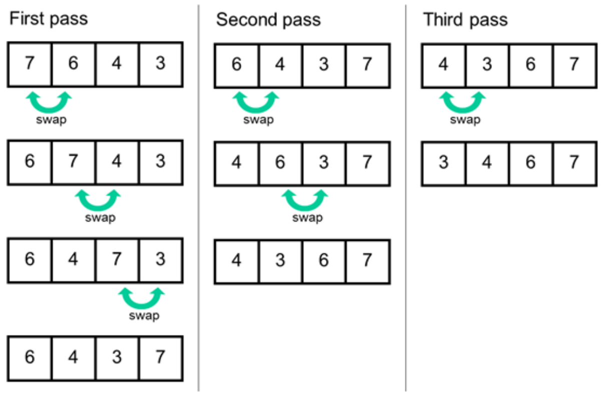

Figure 1.1: Example of Bubble Sort algorithm.

1.2 Time complexity

The Bubble Sort algorithm has a time complexity of $O(N^2)$.

1.3 Space complexity

About space complexity we encounter O(1) since it does not require any additional memory space apart from a temporary variable used for swapping.

1.4 Odd-even sort variant

Odd-even sort is a simple sorting algorithm based on bubble sort, with which it shares some characteristics. It works by comparing all odd/even pairs of elements in a list, and if a pair is in the wrong order (the first element is greater than the second), it swaps its elements. The check continues with adjacent pairs of elements with odd/even positions. The algorithm continues the sorting by alternating between odd/even and even/odd comparisons until the entire list is sorted (Figure 1.2).

1.5 Reasons for the choice

I thought about how I could parallelise bubble sort and the idea I came up with corresponded to the odd-even variant, of which I only later found to be a known type.
This is why I decided to study the new algorithm, discarding the classic bubble sort because of its sequential nature.

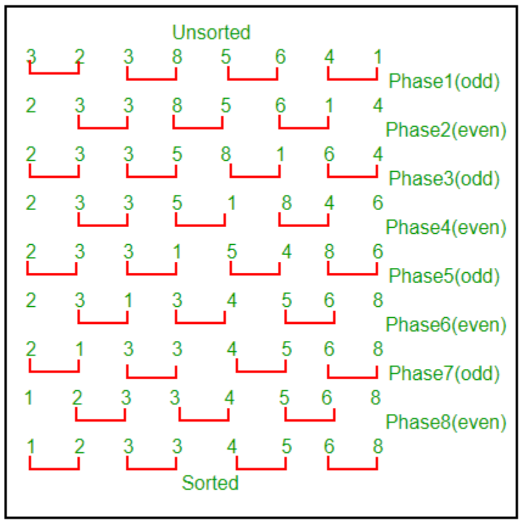

Figure 1.2: Example of Odd-Even Sort algorithm.

1.6 Objectives

Parallelising each iteration by exploiting simultaneous data access, being odd-even acts on data pairs different from each other.

CHAPTER 2
MERGE SORT

2.1 The algorithm

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted, going deep until it reaches the individual units (with the subarrays).

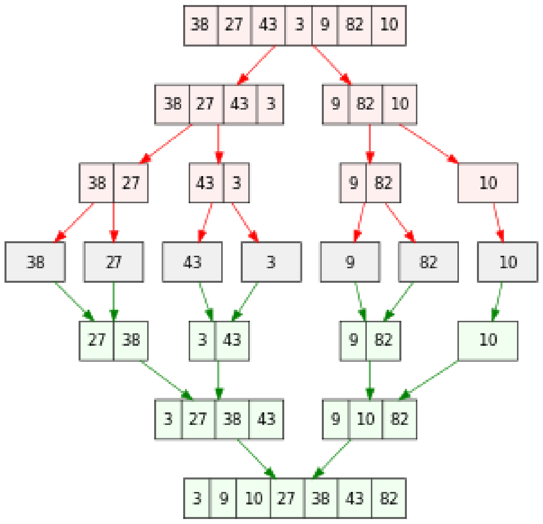

Figure 2.1: Example of Merge Sort algorithm.

2.2 Time complexity

Merge Sort is an efficient, stable sorting algorithm with an average, best-case, and worst-case time complexity of O(nlog n).

2.3 Space complexity

Merge Sort has an additional space complexity of O(n) in its standard implementation.

2.4 Iterative variant

The merge sort thought of in this way has an inherently recursive implementation (for ease of implementation). For this reason, since I had to parallelise, recursion did not help me, whereas the iterative version (from which I was inspired) was more useful. The iterative version simply eliminates the first part focusing on the division into subarrays, starting directly from the individual units. From there it proceeds by doing the progressive merge by pairing the subarrays two by two (Figure 2.2).

2.5 Reasons for the choice

I chose the merge sort to tackle a more elaborate and multi-step algorithm. The algorithm has extremely lower complexity so the challenge was harder.

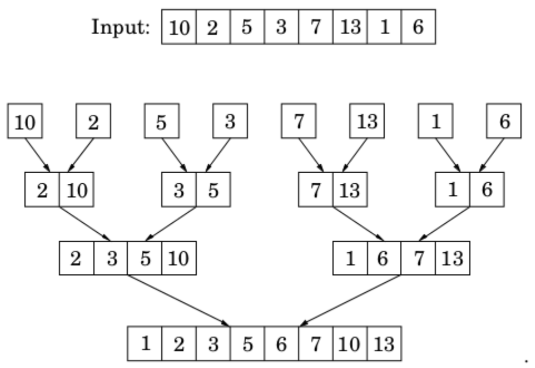

Figure 2.2: Example of Iterative Merge Sort algorithm.

2.6 Objectives

Similarly to the previous idea, I addressed the problem by computing the merge of the subarrays in parallel until the starting array was completely rebuilt, but ordered.

CHAPTER 3
METHODS

In this chapter, I will describe the methods used to exploit parallel programming.

3.1 Odd-Even Sort Parallel Version

As previously written, three versions have been developed for this algorithm in ascending order of completeness, to study how the gain varies as the architecture provided by CUDA is fully exploited.
For all three versions the odd kernel and the even kernel are launched twice for each for loop, which is executed N/2 times (where N is the size of the array). As can be seen in the case of using only blocks, the operations performed match the complexity indicated above. In fact, the for loops are repeated N/2 times (where N is the size of the array and it works with pairs, so there are half operations with respect to N) and N operations are performed by the kernels at each loop.
Furthermore, the loop usually calls N times odd-even, but in this case it is chosen whether to compute the odd or the even branch on the basis of the index. In this parallel implementation the two kernels are both executed at each iteration, so only N/2 calls are sufficient.

3.1.1 Version 1: only blocks runned in parallel

In this particular version only blocks are used to speed up the computation. For each block only one thread is operational.
Due to the fact that there are a lot of iterations the final array could be sorted, but it could happen some numbers are changed because there are multiple access in reading data from memory and nothing synchronising executions.
All the code in this project always work even without synchronisation. I think the reason is that the computational time is very small, so access to memory never overwrite each other. I also tried slowing down some threads with sleep functions or with repeated prints but the result was always a correctly ordered array. I could not figure out whether cpu calls within while or for have default barriers.

3.1.2 Version 2: only blocks plus synchronisation

To stay safe and avoid errors, in the second implementation I used synchronisation barriers between the odd and even branches and before the start of another cycle call. These operations definitely require additional waiting time and slow down the algorithm slightly.

3.1.3 Version 3: threads and blocks runned in parallel

In this version, the entire CUDA architecture is used with a gain in performance. The only noticeable changes are in the kernel calls and the allocation of work units.

3.2 Iterative Merge Sort Parallel Version

For the merge sort, a version was implemented which exploits the architecture consisting of blocks and threads. Overall, the structure is similar or the same as what has already been introduced and explained. The specific function runned on device is "merge parallel", called from the cpu via the global function "mergeSort parallel", which mainly calculates the thread index and the values used to form the subarrays.

The non-use of the synchronisation barrier, which dropped all the advantages gained from parallel execution, needs specific note. The algorithm seems to work, but as mentioned before, it is probably not safe.
Why is there no real gain from using barriers? As an answer I say that the sequential algorithm is already very efficient in itself, so it would take extra work and other constructs to further improve parallel execution. Probably the use of shared memory could be a turning point and also future work to complete the project.

CHAPTER 4
RESULTS AND DISCUSSIONS

The results obtained are shown in the following table 4.1 where the rows indicate the type of algorithm and the columns the various and possible versions. In this way, one can compare the difference in time taken between basic algorithms and their parallel implementations. As mentioned above, we always obtain a gain in computation (starting with the recursive and ending with the implementation with blocks and threads) except for version 2 of the bubble sort where we find the already mentioned overhead due to the introduction of synchronisation.

	Recursive	Iterative	version 1	Version 2	Version 3
Bubble-Sort	7.33	5.69	0.79	0.98	0.29
Merge-Sort	0.0080	0.0073	0.000052		

Table 4.1: results in seconds of the various versions (Please note: the results are indicative to my execution on the computer science and engineering cluster of Unibo.).

We are now ready to calculate the execution time speed up counted as a division between the sequential execution time and the parallel execution time. The result indicates the multiplication factor we have gained with respect to the iterative version which is the best basic time.

	version 1	Version 2	Version 3
Bubble-Sort	7.2x	5.8x	19,6x
Merge-Sort	153x		

Table 4.2: Speed Up.

In the end as can be seen from the results, the parallel versions computed on 50000 elements have a very high speedup factor, which probably scales with the size of the array and increases proportionally to it. These results are due to the simplicity of the task and the fact that we are talking about relatively short time windows, in a more complex problem the speedups are not so high.