Sorting Sequences of Values

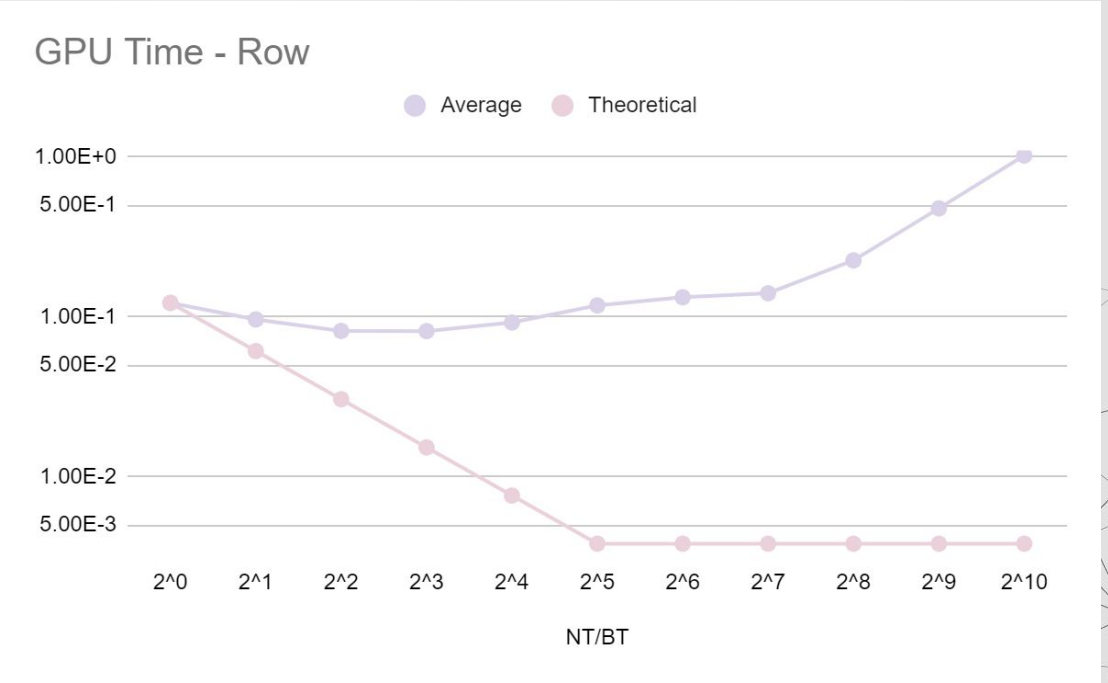
Diogo Correia 90327
Lara Rodrigues 93427

Optimization of launch configuration for Rows

Because the elements of the sequence to be sorted are accessed **sequentially**, the number of **cache misses** is **smaller**, so it's wise to use **more blocks** (greater grid dimension) to take advantage of the GPU parallel processing.

After experimenting with different configurations, the **best results** are achieved when the **block size** is **2^3**. This goes according to what was said above, because it allows for a decent number of blocks (**2^7**).

The block size is the same as the value of a warp (32 threads).

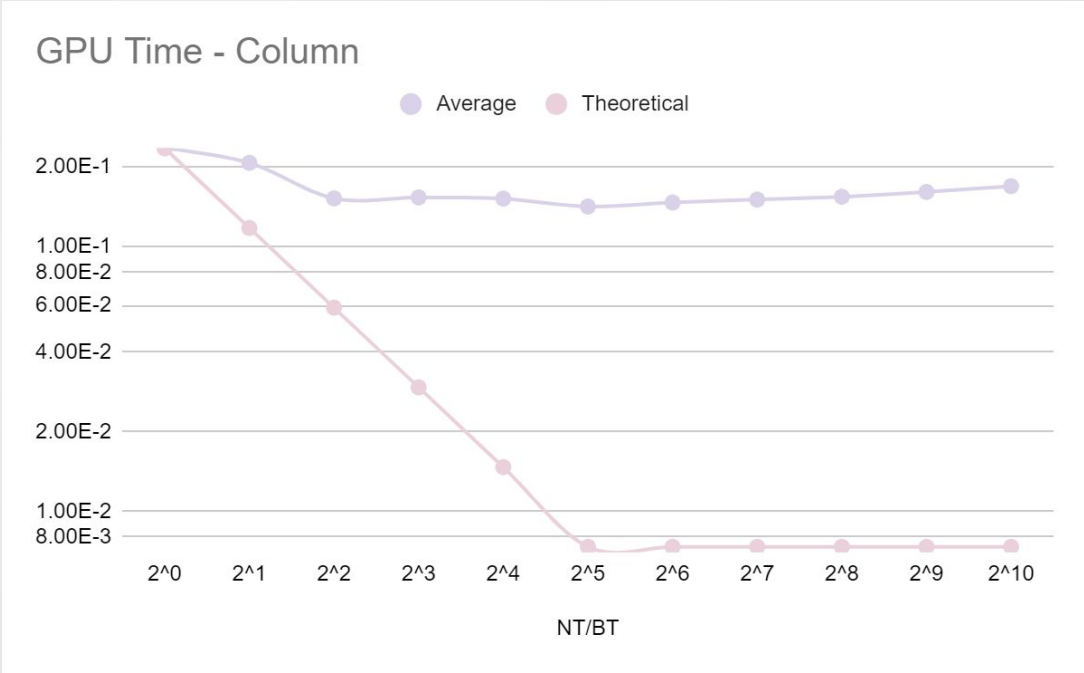


Optimization of launch configuration for Columns

Because the sequences are displayed as **columns**, the next elements will always be 1024 positions ahead. This will **increase** the number of **cache misses**, therefore it is wise to **reduce** the number of **blocks** (less blocks, less memory accesses).

After experimenting with different configurations, the **best results** are achieved when the **block size** is **2^5**. Less number of blocks when compared to sorting rows, but not too low to still take advantage of the GPU parallel processing.

The block size is the same as the value of a warp (32 threads).



Upon the results from the optimization for one dimension, we then froze the best result for the grid size and alternated the values for the block size, and vice versa.

Row config					
gridDimX	blockDimX	gridDimY	blockDimY	Average	Standard Deviation
2^7	2^3	2^0	2^0	8.1556E-02	1.67E-05
2^7	2^2	2^0	2^1	8.1848E-02	3.96E-05
2^7	2^1	2^0	2^2	8.4092E-02	1.30E-05
2^7	2^0	2^0	2^3	8.1790E-02	3.67E-05

Row config					
gridDimX	blockDimX	gridDimY	blockDimY	Average	Standard Deviation
2^7	2^3	2^0	2^0	8.1558E-02	1.48E-05
2^6	2^3	2^1	2^0	8.1546E-02	2.97E-05
2^5	2^3	2^2	2^0	8.1570E-02	2.35E-05
2^4	2^3	2^3	2^0	8.1538E-02	1.92E-05
2^3	2^3	2^4	2^0	8.1542E-02	2.28E-05
2^2	2^3	2^5	2^0	8.1566E-02	1.82E-05
2^1	2^3	2^6	2^0	8.1560E-02	1.22E-05
2^0	2^3	2^7	2^0	8.1598E-02	4.03E-05

Column config					
gridDimX	blockDimX	gridDimY	blockDimY	Average	Standard Deviation
2^5	2^5	2^0	2^0	1.3300E-01	1.13E-02
2^5	2^4	2^0	2^1	1.6030E-01	2.60E-02
2^5	2^3	2^0	2^2	1.5972E-01	2.41E-02
2^5	2^2	2^0	2^3	1.5908E-01	6.12E-03
2^5	2^1	2^0	2^4	1.7270E-01	2.29E-02
2^5	2^0	2^0	2^5	2.0154E-01	3.08E-02

Column config					
gridDimX	blockDimX	gridDimY	blockDimY	Average	Standard Deviation
2^5	2^5	2^0	2^0	1.4326E-01	1.45E-03
2^4	2^5	2^1	2^0	1.4146E-01	1.89E-02
2^3	2^5	2^2	2^0	1.4280E-01	3.96E-03
2^2	2^5	2^3	2^0	1.4224E-01	4.86E-03
2^1	2^5	2^4	2^0	1.4282E-01	4.06E-03
2^0	2^5	2^5	2^0	1.4214E-01	1.07E-02

To understand how good the GPU did over the CPU, we measured the execution times for both cpu kernels (rows and columns) and calculated the **speedup**.

The execution time for the best configuration in the GPU are the following:

- Row Kernel <<<(16,8,1), (8,1,1)>>>: **8.1538E-02**
- Column Kernel <<<(32,1,1), (32,1,1)>>>: **1.3300E-01**

The execution time for CPU, as an average of 5 executions, are the following:

- Row Kernel: **6.4100E-01**
- Column Kernel: **9.3200E+00**

	CPU (Row)	CPU (Columns)
Execution Times	6.468E-01	9.248E+00
	6.414E-01	9.357E+00
	6.414E-01	9.365E+00
	6.337E-01	9.325E+00
	6.416E-01	9.303E+00
Average	6.410E-01	9.320E+00

Speedup for the **Rows**: 6.4100E-01 / 8.1538E-02 = **7.8614**
Speedup for the **Columns**: 9.3200E+00 / 1.3300E-01 = **70.0752**

These results for the speedup show us that, when sorting the sequences by rows, the GPU performed **7.8614** times faster than the CPU. When sorting by columns, the speedup is even greater, of **70.0752**. This was expected, considering all the parallelism the GPU offers, using several blocks of threads, even with all the cache misses that occurred, mainly within the sorting by columns. In conclusion, **it was useful to offload the computation to the GPU**.

Single Sequence with a size of 1024*1024

Because we go from sorting small sequences to sorting one way bigger sequence, it makes more sense to use a different sorting algorithm, that shows better performance for large data sets.

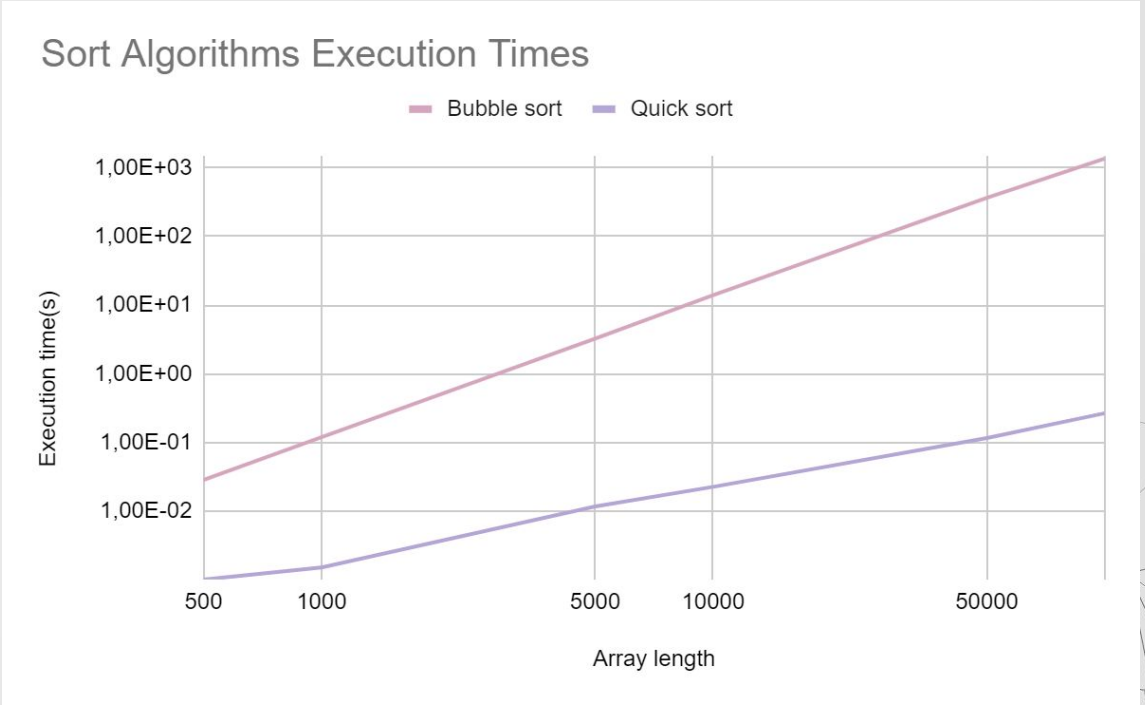
With this in mind, it is better to use Quick Sort, instead of Bubble Sort. Quick Sort is a divide and conquer sorting algorithm that has an average case time complexity of O(n log(n)). On the other hand, bubble sort is a simple sorting algorithm that has a time complexity of O(n^2). This means that the time taken to execute Quick Sort grows at a slower rate than the size of the data being processed.

Time Complexity	Best	Average	Worst
Bubble Sort	O(n)	O(n^2)	O(n^2)
Quick Sort	O(n log(n))	O(n log(n))	O(n^2)

To justify the change in the algorithm, we made a python script that generated several sequences with random values, with different sizes and registered the execution times. The sizes of the sequences tested were 100, 1000, 5000, 10000, 50000 and 100000.
This gives us an idea that **Quick Sort will have a better performance**.

Bubble Sort	
Array length	Execution time(s)
500	2,85E-02
1000	1,19E-01
5000	3,25E+00
10000	1,39E+01
50000	3,62E+02
100000	1,35E+03

Quick sort	
Array length	Execution time(s)
500	1,02E-03
1000	1,52E-03
5000	1,18E-02
10000	2,27E-02
50000	1,16E-01
100000	2,67E-01



To take advantage of the hardware features we have at our disposal, we have decided to go with the **parallel version** of Quick Sort. This version allows the program execution to happen concurrently using multiple processors or threads in order to improve the performance of our program.

Because the algorithm is divide to conquer, it will work as an upside down tree. Upon calculating the pivot for the entire sequence, we apply the sorting mechanism for that pivot, and end up with two new sequences, half the size of the sequence before (one with only numbers smaller than the pivot, and the other only with numbers greater or equal than the pivot). We can apply this method for every subsequent subsequence and we end up with more and more smaller sequences. The sequences that are in the same row of the tree can take advantage of the GPU parallelism because they are independente from each other.

In conclusion, this algorithm allows for:

- Better performance on larger sets
- Concurrent sortings

We end up in a situation where we can use one thread for each subsequence. Then, the **number of threads** in each step of the algorithm is **2^n**, where n = N_ROWS.