Arquiteturas de Alto Desempenho

Project 2 - Sorting Sequences of Values

António Borges

Group 12:
Victor Souza, 89330
Rafael Amorim, 98197

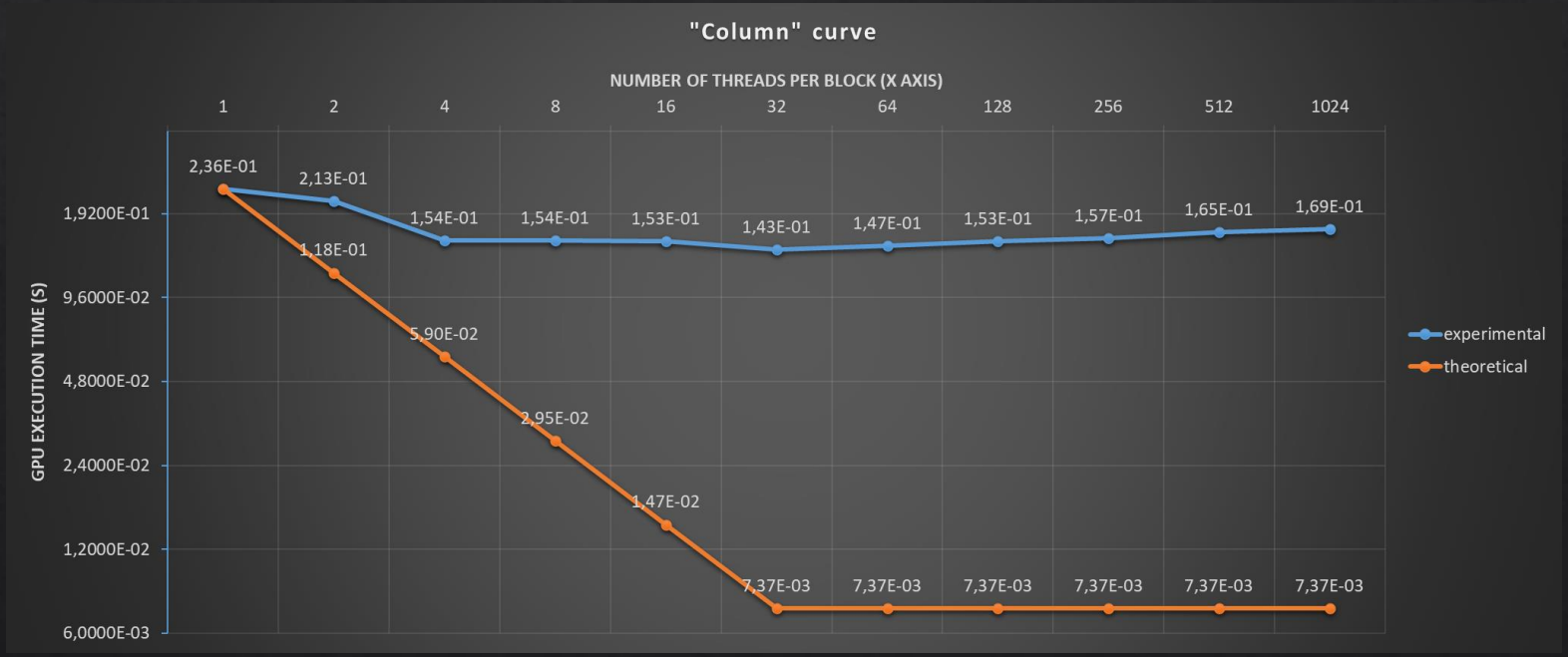
universidade de aveiro

IncOrderColumn

- We started this analysis by varying the number of threads per block on the x-axis (in incOrderColumn.cu) and writing down the respective execution times for the GPU kernel. We also estimated the theoretical values for the execution time.

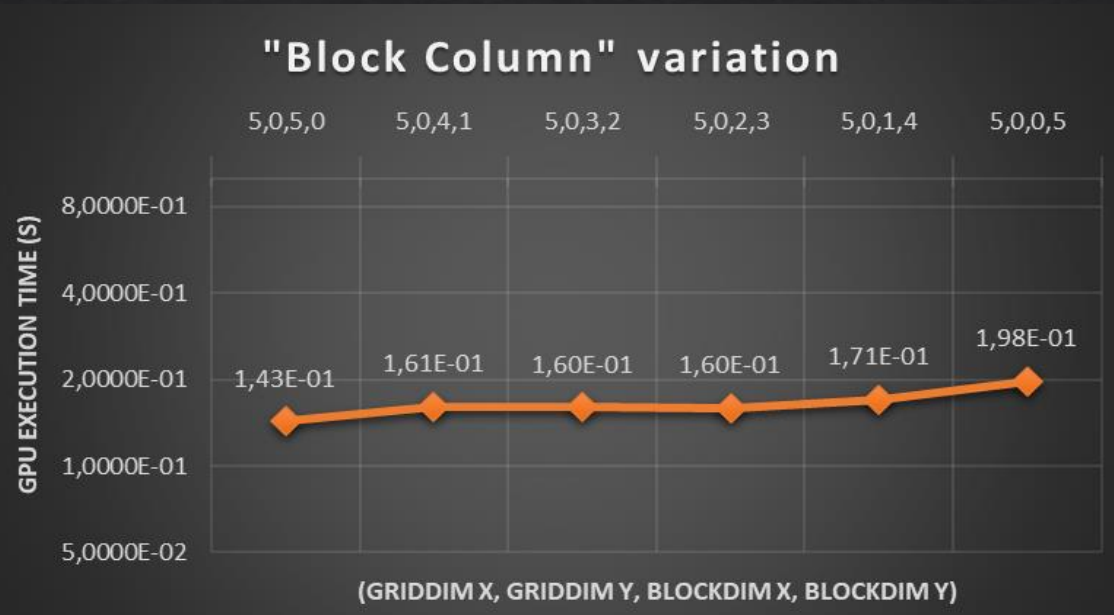
- From looking at it we can conclude that the experimental time doesn't behave in the same way that the theoretical curve suggests it should, in fact, there is very little variation in execution time when increasing the number of threads per block. The biggest time gain happens when the code is executed with 4 threads per block, from there the gains in execution time almost stall, until reaching the **minimum value of 1,43e-01 seconds** with 32 threads per block, at which point the system starts to increase the execution time again. The best value occurring with 32 threads makes sense, since blocks are divided in warps (chunks of 32 threads) when executed in the streaming multiprocessor.

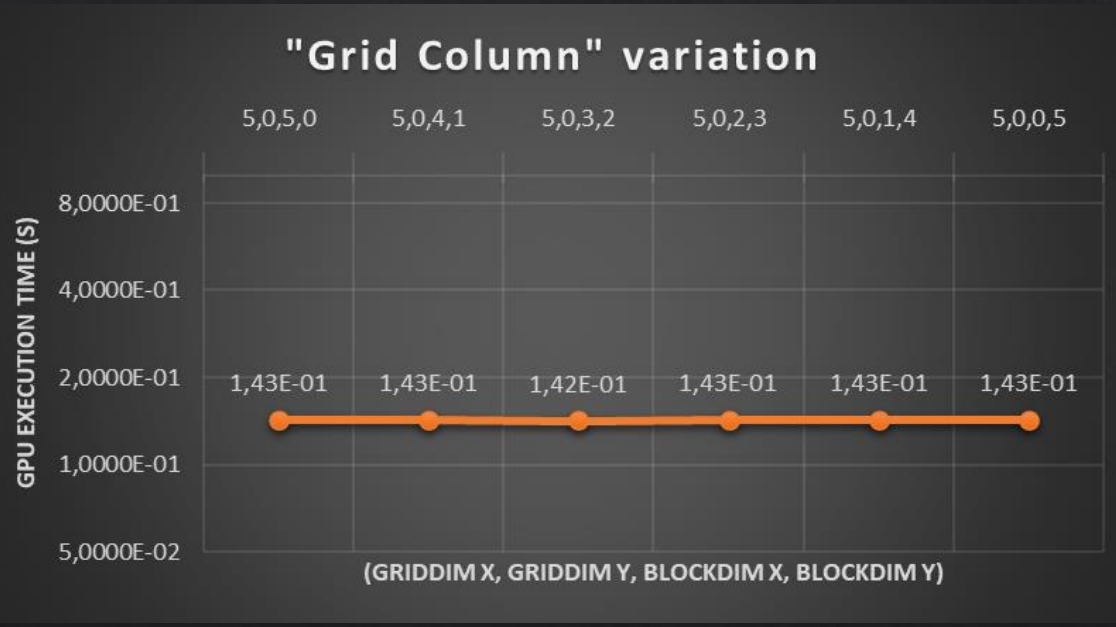
- In this case, every new thread that launches, accesses the next index position of memory, this means that adjacent threads access contiguous memory positions. However, now all elements of a sequence reside far away from each other (at least 1024 positions of distance), which means that every time a thread tries to access a new element of a sequence, there is a high probability that the element isn't in local memory, thus causing miss penalties, that translate to time loss.



IncOrderColumn - Conclusions

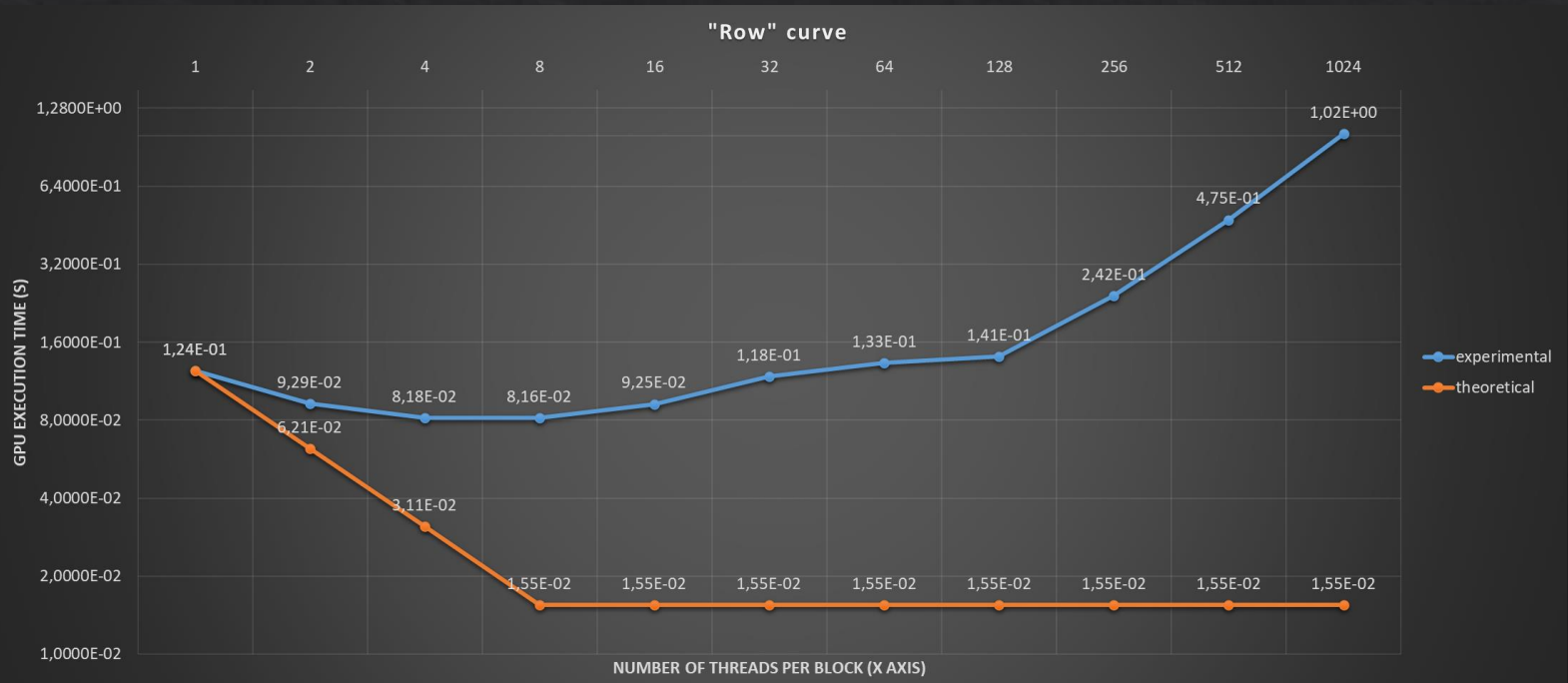
- After that, we confirm the best value in terms of performance. Then, keeping the grid size of minimum time value, i.e., 32 blocks, we varied the values of block's x and y (graph to the left) so that the total sum of exponents would be 5 (2^5 = 32), i.e., 5-0, 4-1, 3-2, 2-3, 1-4, 0-5. The same was done by keeping the blocks size at 32 and varying the grid dimensions (graph to the right).
- In the first case, the conclusion is that the **best case is still the 32 blocks** for the x-axis. In the second case, the values stayed pretty much the same, implying that changing the grid dimensions don't affect the outcome. This makes sense because all blocks are still being executed in the same streaming multiprocessors, so if the number of blocks and their dimensions don't change, the performance shouldn't change either.
- That said, the best configuration is any that has **32 threads per block in the x-axis and 32 blocks per grid** (no matter the distribution of those blocks).
- Conclusion: To verify if solution really has better performance than CPU, **we averaged 5 measurements of CPU, which resulted in 9,46 seconds**. So, using the GPU for this operation is still much more time efficient than using the CPU, because the CPU executes everything sequentially and even though this GPU approach isn't the best, there is still parallelism of instructions and data being done, thus saving a lot of time.





IncOrderRow

- First, we just changed the number of threads per block on the x-axis (in incOrderRow.cu), which resulted in the following graph with execution times for the GPU kernel.
- Therefore, we understand that the execution time decreases until reaching 8 threads per block, from here we get the best value, then the time increases slightly until reaching 128 threads per block and finally the time gets much worse from 256 to 1024 threads per block.
- Inefficient access arises because of the way that the data is organized in this case. Threads that are sequential don't access contiguous memory sections because they each are placed at a "sequence" of distance (1024 memory positions), because every thread starts at idx*1024. Meaning that there will be many miss situations in the block shared memory.
- While there is some benefit to running more threads in parallel from 2 to 8 threads per block, this benefit is offset by the negative impact on memory transactions and cache misses when using 16 to 1024 threads per block.
- It is also worth noting that once the number of threads per block reaches 64 (resulting in 16 blocks), full GPU occupancy is no longer possible due to the limited number of SMs available on the GPU.



IncOrderRow - Conclusions

- After that, we confirm the best value in terms of performance. Then, keeping the grid value at the point where the time value is minimum value, i.e., 128 blocks per grid (2^7), we varied the values of grid's x and y (graph to the left) so that the total sum exponents would be 3 (2^3 = 8), i.e., 3-0, 2-1, 1-2, 0-3. The same was done by keeping the blocks at 8 threads per block and varying the grid dimensions (graph to the right).
- It turned out that the difference was significantly minimal between this variation. The **best result after this configuration** is any that has **8 threads per block in the x-axis and 128 blocks per grid**. (no matter the distribution of those blocks)
- Conclusion: To verify if this solution really has better performance than the CPU, **we averaged 5 measurements of the CPU, which resulted in 6.2572e-01 seconds**, so despite not being the best solution, it can have better performance than the sequential version of the CPU.