universidade
de aveiro

Arquitecturas de Alto Desempenho

Assignment 2 – Sorting Sequences of Values / GPU Threading and Memory Mapping

Grupo 1, Lab 3

Lucas Pinto, n.º 98500
Carlos Vidal, n.º 23238

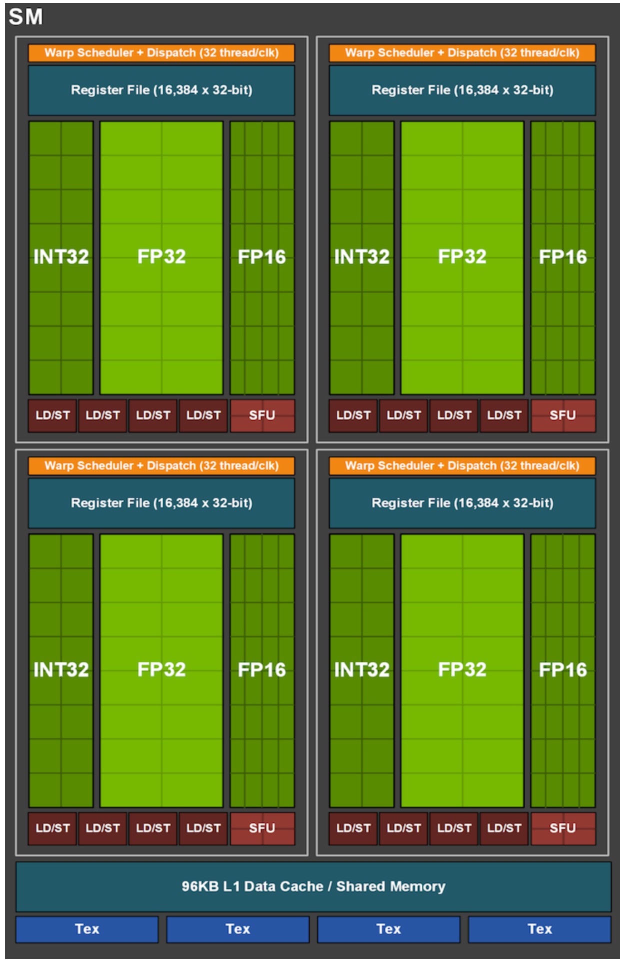
The aim of the assignment is to sort sequences of values on two versions of the bubble sort algorithm on a CPU and a GPU.

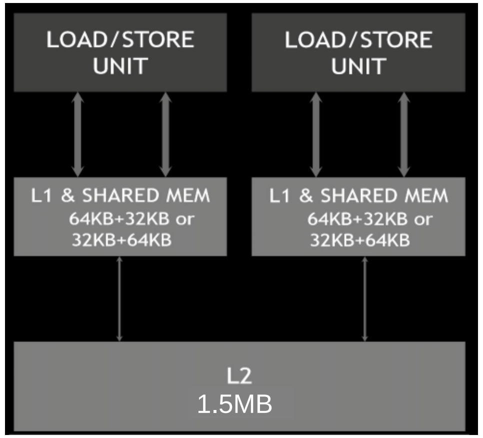
For analising the mapping between each running thread and the memory region it accesses, we studied the GPU arquitecture and organizationof of the assigned GPU board:

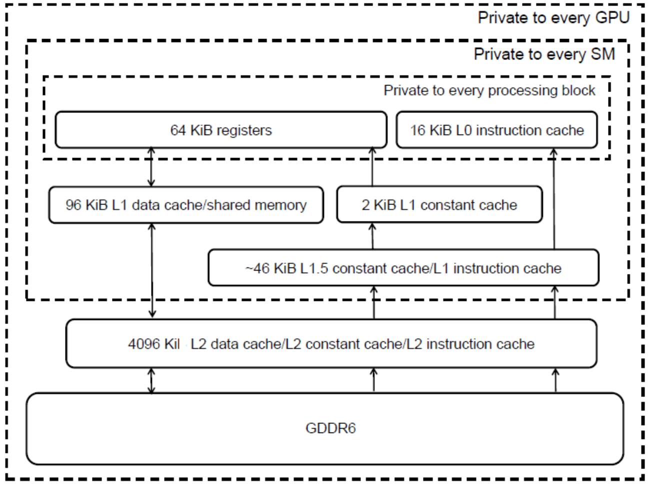
```
/usr/local/cuda-12.0/samples/1_Utilities/deviceQuery$ ./deviceQuery
```

NVIDIA Device: GeForce GTX 1660 Ti

STREAMING MULTIPROCESSOR (SM) ARCHITECTURE







Memory hierarchy of the Turing T4 GPU (TU104).

GeForce GTX 1660 Ti Main Specification		
GPU Engine Specs:	Architecture	Turing TU116-400-A1 Chip
	Compute Capability	7.5
	Transistor Count	6.6x10^9
	SM Count	24
	NVIDIA CUDA® Cores	1536 (24) Multiprocessors, (64) CUDA Cores/MP
	Boost Clock (GHz)	1770
	Base Clock (GHz)	1500
Memory Specs:	Tex L1 Cache	32 KB per SM
	L1 Cache	64 KB per SM
	L2 Cache	1536 KB
	Standard Memory Config	5945 MBytes (6233391104 bytes) GDDR6
	Memory Interface Width	192-bit
	Total amount of shared memory per Block	49152 bytes
	Total number of registers available per Block	65536
	Maximum number of resident blocks per SM	16
	Maximum number of resident warps per SM	32
	Maximum number of threads per Block	1024
	Maximum number of resident threads per SM	1024
	Max dimension size of a thread block (x,y,z)	(1024, 1024, 64)
	Max dimension size of a grid size (x,y,z)	(2147483647, 65535, 65535)
	Maximum memory pitch	2147483647 bytes

The Bubble Sort Algorithm

On one version, the elements are seen as forming the rows of a matrix, each row corresponding to a different sequence to be sorted.
In the second, the elements are seen as forming the columns of a matrix, each column corresponding to a different sequence to be sorted.

```
#ifndef ARRAY_LENGTH
# define ARRAY_LENGTH  (1 << 10)            //(2^10=1024)
#endif
#ifndef N_ARRAYS
# define N_ARRAYS  (1 << 10)                //(2^10=1024)
#endif

data_size = (size_t) N_ARRAYS * (size_t) ARRAY_LENGTH * sizeof (unsigned int);
```

Data_size = $2^{10} * 2^{10} * 2^2$ bytes = 2^{22} bytes = 4 MB (4 194 304 bytes) <=>
$2^{22} * 2^3$ bits = 2^{25} bits = 33 554 432 bits

TreadID and launch configuration, running both versions with various blockDimX and blockDimY configurations, from 2^0 to 2^{10}, comparing the average time execution, using the best Block performance to test various Grid X and Y configurations.

```
blockDimX = 1 << 0;
blockDimY = 1 << 0;
blockDimZ = 1 << 0;
gridDimX = 1 << 10;            //(2^10=1024)
gridDimY = 1 << 0;
gridDimZ = 1 << 0;

x = (unsigned int) threadIdx.x + (unsigned int) blockDim.x * (unsigned int) blockIdx.x;
y = (unsigned int) threadIdx.y + (unsigned int) blockDim.y * (unsigned int) blockIdx.y;
idx = (unsigned int) blockDim.x * (unsigned int) gridDim.x * y + x;
```

Row Sorting algorithm

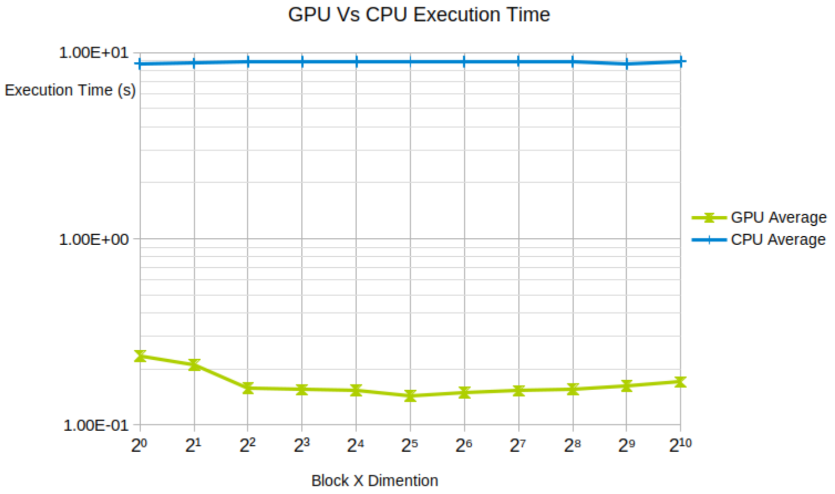
```
data += length * idx; // adjust pointer to the array to be ordered

for (i = 0; i < length - 1; i++)
    { noSwap = true;
        for (j = length - 1; j > i; j--)
        if (data[j] < data[j-1])
            { tmp = data[j];
            data[j] = data[j-1];
            data[j-1] = tmp;
            noSwap = false;
            }
        if (noSwap) break;
    }
```

Column Sorting algorithm

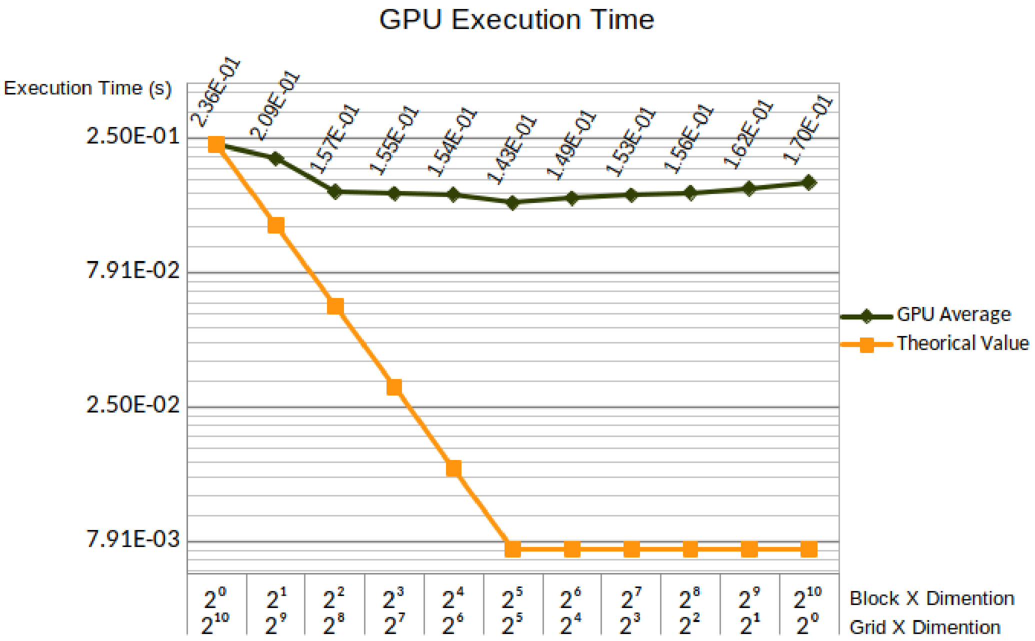
```
data += idx; // adjust pointer to the array to be ordered

for (i = 0; i < length - 1; i++)
    { noSwap = true;
        for (j = length - 1; j > i; j--)
        if (data[j*N_ARRAYS] < data[(j-1)*N_ARRAYS])
          { tmp = data[j*N_ARRAYS];
          data[j*N_ARRAYS] = data[(j-1)*N_ARRAYS];
          data[(j-1)*N_ARRAYS] = tmp;
          noSwap = false;
          }
        if (noSwap) break;
    }
```

Time Execution Results for The Row Sorting

GPU													CPU
Grid		Block		Time (s)									
x	y	x	y	1st Run	2nd Run	3rd Run	4th Run	5th Run	GPU Average	Standard Deviation	SD/Ave	Theorical Value	CPU Average
2^{10}	2^0	2^0	2^0	2.35E-01	2.35E-01	2.34E-01	2.37E-01	2.39E-01	2.359E-01	1.67E-03	0.71%	2.36E-01	8.70E+00
2^9	2^0	2^1	2^0	2.08E-01	2.11E-01	2.09E-01	2.07E-01	2.11E-01	2.091E-01	1.72E-03	0.82%	1.18E-01	8.73E+00
2^8	2^0	2^2	2^0	1.56E-01	1.56E-01	1.60E-01	1.56E-01	1.57E-01	1.570E-01	1.42E-03	0.91%	5.90E-02	8.91E+00
2^7	2^0	2^3	2^0	1.55E-01	1.55E-01	1.54E-01	1.53E-01	1.57E-01	1.547E-01	1.16E-03	0.75%	2.95E-02	8.86E+00
2^6	2^0	2^4	2^0	1.54E-01	1.54E-01	1.54E-01	1.54E-01	1.53E-01	1.539E-01	3.89E-04	0.25%	1.47E-02	8.88E+00
2^5	2^0	2^5	2^0	1.44E-01	1.44E-01	1.41E-01	1.43E-01	1.42E-01	1.430E-01	1.19E-03	0.83%	7.37E-03	8.91E+00
2^4	2^0	2^6	2^0	1.50E-01	1.49E-01	1.48E-01	1.49E-01	1.48E-01	1.487E-01	7.05E-04	0.47%	7.37E-03	8.94E+00
2^3	2^0	2^7	2^0	1.52E-01	1.51E-01	1.55E-01	1.53E-01	1.51E-01	1.526E-01	1.57E-03	1.03%	7.37E-03	8.96E+00
2^2	2^0	2^8	2^0	1.56E-01	1.56E-01	1.56E-01	1.56E-01	1.57E-01	1.563E-01	1.51E-04	0.10%	7.37E-03	8.87E+00
2^1	2^0	2^9	2^0	1.62E-01	1.64E-01	1.62E-01	1.62E-01	1.62E-01	1.622E-01	8.70E-04	0.54%	7.37E-03	8.71E+00
2^0	2^0	2^{10}	2^0	1.70E-01	1.70E-01	1.69E-01	1.71E-01	1.71E-01	1.701E-01	6.29E-04	0.37%	7.37E-03	8.94E+00
2^4	2^1	2^5	2^0	1.43E-01	1.44E-01	1.44E-01	1.44E-01	1.43E-01	1.434E-01	6.27E-04	0.44%		

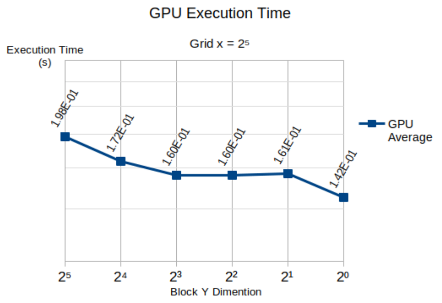


-

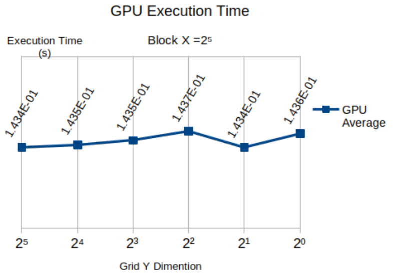


The average relative speedup execution time for the CPU and the best GPU configuration is $S = \frac{8{,}91x10^0}{1{,}43x10^{-1}} = 62{,}30 \Leftrightarrow 6.230\%$

GPU											
Grid		Block									
x	y	x	y	1st Run	2nd Run	3rd Run	4th Run	5th Run	GPU	Standard	SD/Ave
2^5	0	2^0	2^5	1,97E-01	1,98E-01	1,98E-01	1,98E-01	1,98E-01	1,98E-01	5,17E-04	0,26%
2^5	0	2^1	2^4	1,71E-01	1,73E-01	1,73E-01	1,73E-01	1,72E-01	1,72E-01	5,50E-04	0,32%
2^5	0	2^2	2^3	1,60E-01	1,61E-01	1,61E-01	1,59E-01	1,59E-01	1,60E-01	9,35E-04	0,58%
2^5	0	2^3	2^2	1,60E-01	1,60E-01	1,61E-01	1,60E-01	1,58E-01	1,60E-01	1,17E-03	0,73%
2^5	0	2^4	2^1	1,61E-01	1,61E-01	1,61E-01	1,61E-01	1,61E-01	1,61E-01	2,50E-04	0,15%
2^5	0	2^5	2^0	1,43E-01	1,43E-01	1,40E-01	1,42E-01	1,40E-01	1,42E-01	1,26E-03	0,89%

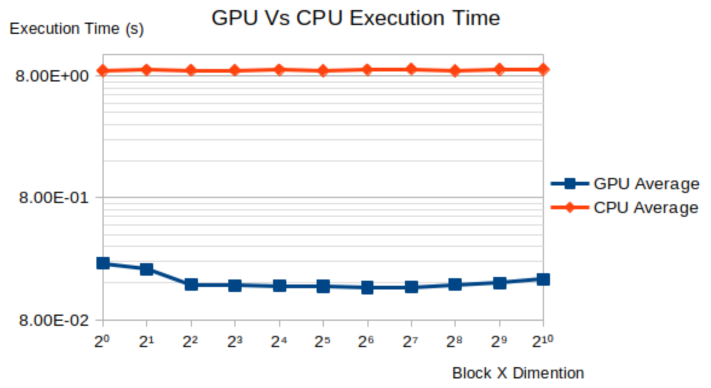


GPU											
Grid		Block									
x	y	x	y	1st Run	2nd Run	3rd Run	4th Run	5th Run	GPU	Standard	SD/Ave
2^0	2^5	2^5	2^0	1,44E-01	1,44E-01	1,41E-01	1,44E-01	1,44E-01	1,434E-01	1,14E-03	0,79%
2^1	2^4	2^5	2^0	1,44E-01	1,44E-01	1,43E-01	1,42E-01	1,44E-01	1,435E-01	6,01E-04	0,42%
2^2	2^3	2^5	2^0	1,44E-01	1,43E-01	1,42E-01	1,44E-01	1,44E-01	1,435E-01	9,02E-04	0,63%
2^3	2^2	2^5	2^0	1,43E-01	1,44E-01	1,43E-01	1,44E-01	1,44E-01	1,437E-01	5,71E-04	0,40%
2^4	2^1	2^5	2^0	1,43E-01	1,44E-01	1,44E-01	1,44E-01	1,43E-01	1,434E-01	6,27E-04	0,44%
2^5	2^0	2^5	2^0	1,44E-01	1,43E-01	1,44E-01	1,43E-01	1,44E-01	1,436E-01	6,61E-04	0,46%

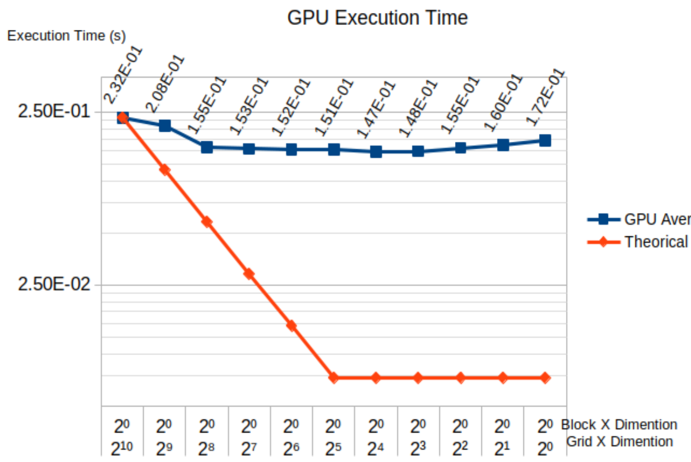


Time Execution Results for The Column Sorting

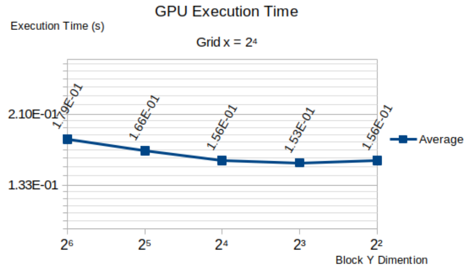
GPU													CPU
Grid		Block		Time (s)									
x	y	x	y	1st Run	2nd Run	3rd Run	4th Run	5th Run	GPU Average	Standard Deviation	SD/Ave	Theorical Value	CPU Average
2^{10}	2^0	2^0	2^0	2,31E-01	2,34E-01	2,33E-01	2,34E-01	2,28E-01	2,32E-01	2,78E-03	1,20%	2,32E-01	8,85E+00
2^9	2^0	2^1	2^0	2,10E-01	2,07E-01	2,07E-01	2,07E-01	2,09E-01	2,08E-01	1,17E-03	0,56%	1,16E-01	8,97E+00
2^8	2^0	2^2	2^0	1,56E-01	1,56E-01	1,55E-01	1,55E-01	1,56E-01	1,55E-01	4,34E-04	0,28%	5,80E-02	8,92E+00
2^7	2^0	2^3	2^0	1,52E-01	1,53E-01	1,53E-01	1,53E-01	1,53E-01	1,53E-01	3,16E-04	0,21%	2,90E-02	8,93E+00
2^6	2^0	2^4	2^0	1,51E-01	1,51E-01	1,52E-01	1,53E-01	1,52E-01	1,52E-01	7,84E-04	0,52%	1,45E-02	8,98E+00
2^5	2^0	2^5	2^0	1,54E-01	1,46E-01	1,52E-01	1,51E-01	1,52E-01	1,51E-01	3,06E-03	2,03%	7,25E-03	8,85E+00
2^4	2^0	2^6	2^0	1,47E-01	1,44E-01	1,49E-01	1,47E-01	1,47E-01	1,47E-01	1,78E-03	1,21%	7,25E-03	9,00E+00
2^3	2^0	2^7	2^0	1,51E-01	1,48E-01	1,43E-01	1,50E-01	1,50E-01	1,48E-01	3,36E-03	2,26%	7,25E-03	9,13E+00
2^2	2^0	2^8	2^0	1,54E-01	1,56E-01	1,57E-01	1,54E-01	1,56E-01	1,55E-01	1,35E-03	0,87%	7,25E-03	8,83E+00
2^1	2^0	2^9	2^0	1,61E-01	1,66E-01	1,60E-01	1,61E-01	1,56E-01	1,60E-01	3,53E-03	2,20%	7,25E-03	9,06E+00
2^0	2^0	2^{10}	2^0	1,76E-01	1,68E-01	1,69E-01	1,70E-01	1,76E-01	1,72E-01	4,01E-03	2,34%	7,25E-03	9,10E+00



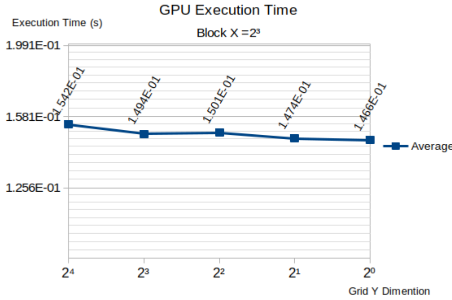
The average relative speedup execution time for the best GPU configuration, against the CPU, is $S = \frac{8{,}970x10^{0}}{1{,}466x10^{-1}}$ = 61,19 ⇔ 6.119%, wich clearly shows the advantage of using the GPU.



GPU											
Grid		Block									
x	y	x	y	1st Run	2nd Run	3rd Run	4th Run	5th Run	Average	Standard Deviation	SD/Ave
2^4	2^0	2^0	2^6	1,80E-01	1,79E-01	1,79E-01	1,78E-01	1,78E-01	1,79E-01	1,03E-03	0,57%
2^4	2^0	2^1	2^5	1,66E-01	1,66E-01	1,66E-01	1,68E-01	1,65E-01	1,66E-01	1,11E-03	0,67%
2^4	2^0	2^2	2^4	1,55E-01	1,56E-01	1,56E-01	1,55E-01	1,58E-01	1,56E-01	1,29E-03	0,83%
2^4	2^0	2^3	2^3	1,52E-01	1,53E-01	1,52E-01	1,54E-01	1,57E-01	1,53E-01	2,25E-03	1,47%
2^4	2^0	2^4	2^2	1,53E-01	1,55E-01	1,57E-01	1,55E-01	1,59E-01	1,56E-01	2,20E-03	1,42%



GPU											
Grid		Block									
x	y	x	y	1st Run	2nd Run	3rd Run	4th Run	5th Run	Average	Standard Deviation	SD/Ave
2^4	2^0	2^3	2^3	1,54E-01	1,53E-01	1,55E-01	1,54E-01	1,56E-01	1,542E-01	9,63E-04	0,62%
2^3	2^1	2^3	2^3	1,49E-01	1,50E-01	1,51E-01	1,48E-01	1,50E-01	1,494E-01	1,01E-03	0,67%
2^2	2^2	2^3	2^3	1,50E-01	1,49E-01	1,51E-01	1,49E-01	1,51E-01	1,501E-01	1,29E-03	0,86%
2^1	2^3	2^3	2^3	1,48E-01	1,48E-01	1,48E-01	1,46E-01	1,48E-01	1,474E-01	7,33E-04	0,50%
2^0	2^4	2^3	2^3	1,46E-01	1,46E-01	1,45E-01	1,47E-01	1,49E-01	1,466E-01	1,22E-03	0,83%



Conclusions:

On both cases, the speedup gains resulting from the multithreading on the block axis is far from what was expected on theory.
Up to 32 (2^5) threads per block, a speed up of 2 (200%) was espected each time the number of threads per block doubles, wich did not verified.
This can be explained with the poor mapping between each specific thread and the values in rows or columns to be read and written, causing data hazards.
Scatter operations on graphics processors may cause write-after-a-read hazards between multiple fragment processors accessing the same memory location. Therefore, most sorting algorithms cannot be efficiently implemented on GPUs.

On the second part of the assignment we were asked to sketch how the program that was given could be changed, if instead of having multiple sequences to be sorted independently, we would sort the total data comprised of 1024x1024 4byte elements (as shown on slide n.º 2).

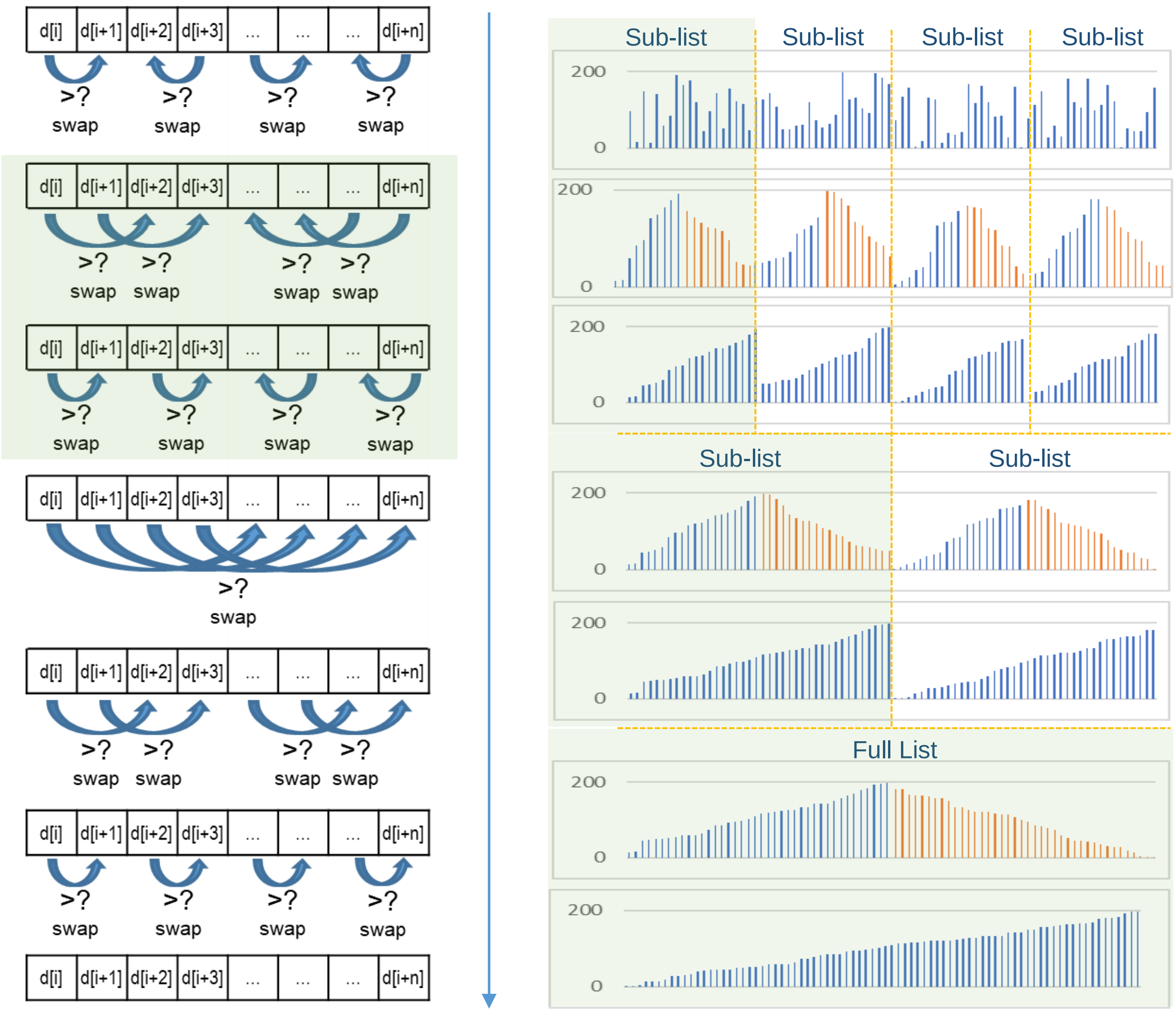
On that matter, ussing the same strategy of the bubble sort algoritm on the GPU was not plausible, because of the sequential nature of the algoritm, that prevents it from paralelizing the sorting operation as a whole.

For that, we studied other aplicable algoritms. At best we decided to presente three potentialy good candidates: the bitonic sort algorithm, the odd-even sort, and a hibrid aplication of the former bubble sort.

Bitonic Sort Algorithm:

- For a N x N matrix, the number of the initial sub-list would be $\sqrt{NxN} = N$.
- Each (sub)list sorting is comprised of 3 stages
- Being the number of elements to be ordered “m”, the number of independent threads for Stage 1 would be m/2, having only one pass.
- For Stage 2, the number of independent threads would also be m/2, but this fase needs 2 passes.
- For Stage 3, the number of independent threads would once more be m/2, with 3 passes nedded.
- Total space complexity = $O(n.\log^2 n)$. For n=1024x1024 ⇔ $1024^2 x(\log 1024^2)^2$ ⇔ $1024^2 x(2\log 1024)^2 = 38{,}008x10^6$ comparisons.
- Total time complexity is $O(\log^2 n)$. For $n=1024^2$ ⇔ $(\log 1024^2)^2 = (2\log 1024)^2 = 3{,}625x10^1$
- The time complexity of the buble sort is for the worst case is $O(n^2)$, runed in sequencial, wich is worst than the basic bitonic sort.

Bitonic Sort Algorithm

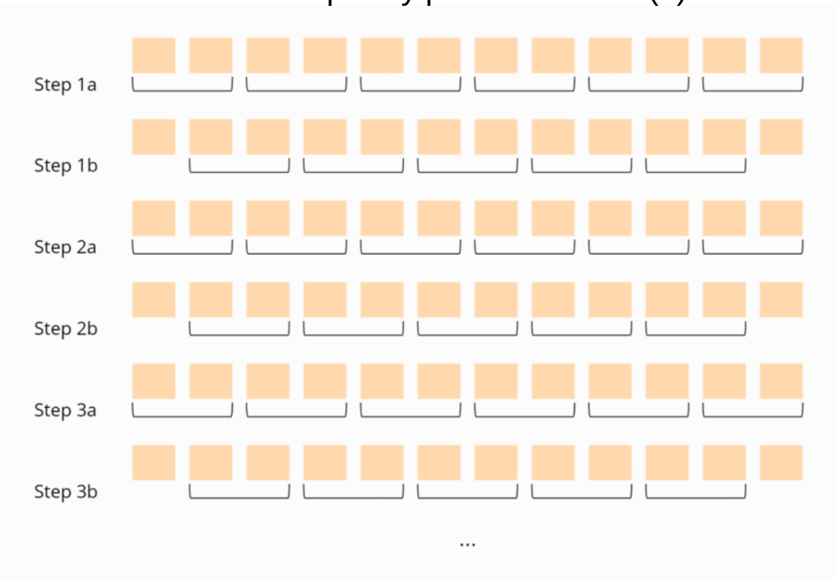


Odd-Even Sort

Compare in parallel the first with the second element, the third with the fourth, the fifth with the sixth, etc. and swap the respective elements if the left one is larger than the right one.

Then you compare the second element with the third, the fourth with the fifth, the sixth with the seventh, and so on.
These two steps are alternated until no more elements are swapped in either step.

Worst case time complexity parallelized = O(n):

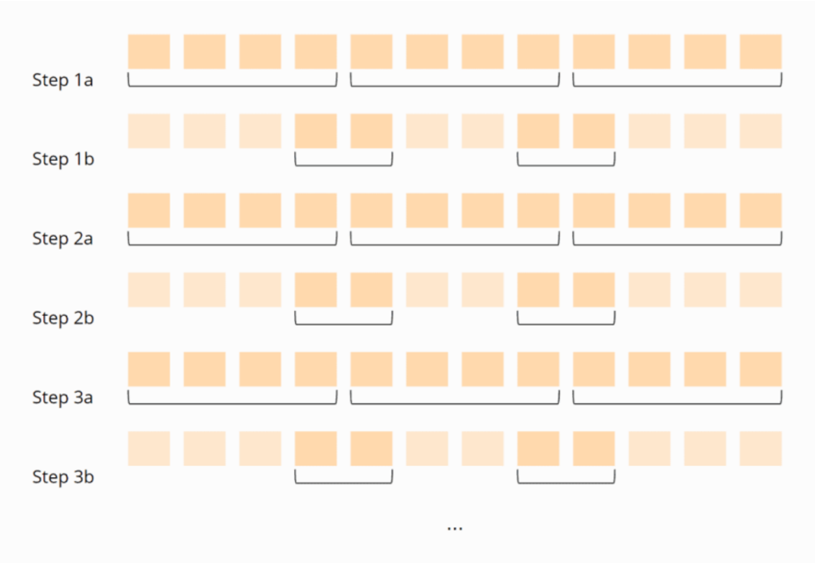


Hibrid Aplication of The Bubble Sort Algorithm

First, the whole unsorted sequence is divided to best map to memory [1].

Now one Bubble Sort iteration is performed in all partitions in parallel. Wait until all threads are finished, and then compare the last element of one partition with the first of the next partition. When all threads are finished, the process starts again.

These steps are repeated until no more elements are swapped in all threads:



[1] The number of steps a kernel can cover is bounded by the number of items that is possible to be included in the stream element. Furthermore, the number of items is bounded by the size of the shared memory available for each SIMD processor. If each SM has 16 KB of local memory, then we can specify a partition consisting of SH = 4K items, for 32- bit items. (32bits×4K = 15.6KB) Moreover such a partition is able to cover “at least” sh = lg(SH) = 12 steps (because we assume items within the kernel are only compared once).
If a partition representing an element of the stream contains SH items, and the array to sort contains N = 2n items, then the stream contains b = N/SH = 2n−sh elements. The first kernel can compute (sh)×(sh+1) 2 steps because our conservative assumption is that each number is only compare to one other number in the element. In fact, bitonic sort is structured so that many local comparisons happen at the start of the algorithm.

Sorting with GPUs: A Survey, Dmitri I. Arkhipov, Di Wu, Keqin Li, and Amelia C. Regan. 8 Sep. 2017, on https://arxiv.org/pdf/1709.02520.pdf