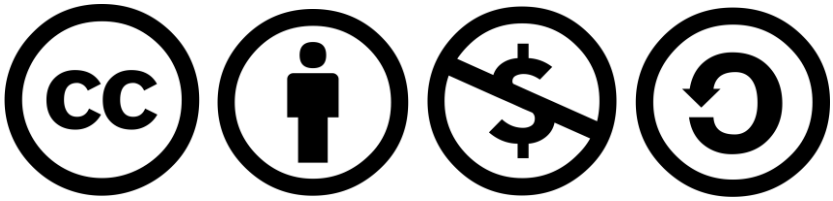


Report Common Assignment CUDA01

Parallel Radix Sort

Lecturer: Francesco Moscato - fmoscato@unisa.it

Group:

- **Marseglia Mattia 0622701697 m.marseglia1@studenti.unisa.it**
- **Spingola Camilla 0622701698 c.spingola@studenti.unisa.it**
- **Sica Ferdinando 0622701794 f.sica24@studenti.unisa.it**
- **Turi Vito 0622701795 v.turi3@studenti.unisa.it**

Index

Problem description

Parallelize and Evaluate Performances of "RADIX SORT" Algorithm, by using CUDA.

Provide (different if you WANT) solutions to the problem of parallelize Radix Sort Algorithm.

Analyze results with different memory allocation (Global, Texture, Shared).

Get execution times, bandwidth and other performances measures when in the different configurations.

Evaluate performances with different block-grid configurations.

Evaluate performances with or without the use of streams.

Experimental setup

Hardware

CPU

```
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 63
model name  : Intel(R) Xeon(R) CPU @ 2.30GHz
stepping    : 0
microcode   : 0x1
cpu MHz           : 2299.998
cache size  : 46080 KB
physical id : 0
siblings    : 2
core id           : 0
cpu cores   : 1
apicid            : 0
initial apicid    : 0
fpu         : yes
fpu_exception     : yes
cpuid level : 13
wp          : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb
rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid
tsc_known_freq pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic
movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm
invpcid_single ssbd ibrs ibpb stibp fsgsbase tsc_adjust bmi1 avx2 smep
bmi2 erms invpcid xsaveopt arat md_clear arch_capabilities
bugs        : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf
mds swapgs
bogomips    : 4599.99
clflush size      : 64
cache_alignment   : 64
address sizes     : 46 bits physical, 48 bits virtual

processor   : 1
vendor_id   : GenuineIntel
cpu family  : 6
model       : 63
model name  : Intel(R) Xeon(R) CPU @ 2.30GHz
stepping    : 0
microcode   : 0x1
cpu MHz           : 2299.998
cache size  : 46080 KB
physical id : 0
siblings    : 2
core id           : 0
```

```
cpu cores   : 1
apicid          : 1
initial apicid   : 1
fpu         : yes
fpu_exception    : yes
cpuid level : 13
wp          : yes
flags       : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca
cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb
rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid
tsc_known_freq pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic
movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm
invpcid_single ssbd ibrs ibpb stibp fsgsbase tsc_adjust bmi1 avx2 smep
bmi2 erms invpcid xsaveopt arat md_clear arch_capabilities
bugs        : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf
mds swapgs
bogomips    : 4599.99
clflush size     : 64
cache_alignment  : 64
address sizes    : 46 bits physical, 48 bits virtual
```

GPU

```
Device name: Tesla K80
  Compute capability: 3.7

  Clock Rate: 823500 kHz
  Total SMs: 13
  Shared Memory Per SM: 114688 bytes
  Registers Per SM: 131072 32-bit
  Max threads per SM: 2048
  L2 Cache Size: 1572864 bytes
  Total Global Memory: 11996954624 bytes
  Memory Clock Rate: 2505000 kHz

  Max threads per block: 1024
  Max threads in X-dimension of block: 1024
  Max threads in Y-dimension of block: 1024
  Max threads in Z-dimension of block: 64

  Max blocks in X-dimension of grid: 2147483647
  Max blocks in Y-dimension of grid: 65535
  Max blocks in Z-dimension of grid: 65535

  Shared Memory Per Block: 49152 bytes
  Registers Per Block: 65536 32-bit
  Warp size: 32
```

- The Tesla K80 has a maximum of 16 blocks for SM and one SM can have a maximum 2048 threads.

Bandwidth

```
[CUDA Bandwidth Test] - Starting...
Running on...

 Device 0: Tesla K80
 Range Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)     Bandwidth(MB/s)
   1000                      226.5
   101000                    6164.2
   201000                    6042.3
   301000                    6754.5
   401000                    6644.6
   501000                    7069.0
   601000                    7049.5
   701000                    6789.3
   801000                    6861.6
   901000                    7257.1

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)     Bandwidth(MB/s)
   1000                      391.0
   101000                    6338.0
   201000                    7012.2
   301000                    7154.8
   401000                    7343.5
   501000                    7446.7
   601000                    7417.3
   701000                    7498.5
   801000                    7569.6
   901000                    7532.5

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)     Bandwidth(MB/s)
   1000                      290.2
   101000                    28505.3
   201000                    44681.0
   301000                    56468.9
   401000                    75111.5
   501000                    82057.2
   601000                    90651.5
   701000                    100322.8
   801000                    91253.7
   901000                    80163.1
```

Software

Google Colab
Colaboratory, or “Colab” for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education. More technically, Colab is a hosted Jupyter notebook service that requires no setup to use, while providing free access to computing resources including GPUs.

Performance

Case n° 1 – Without CUDA Streams

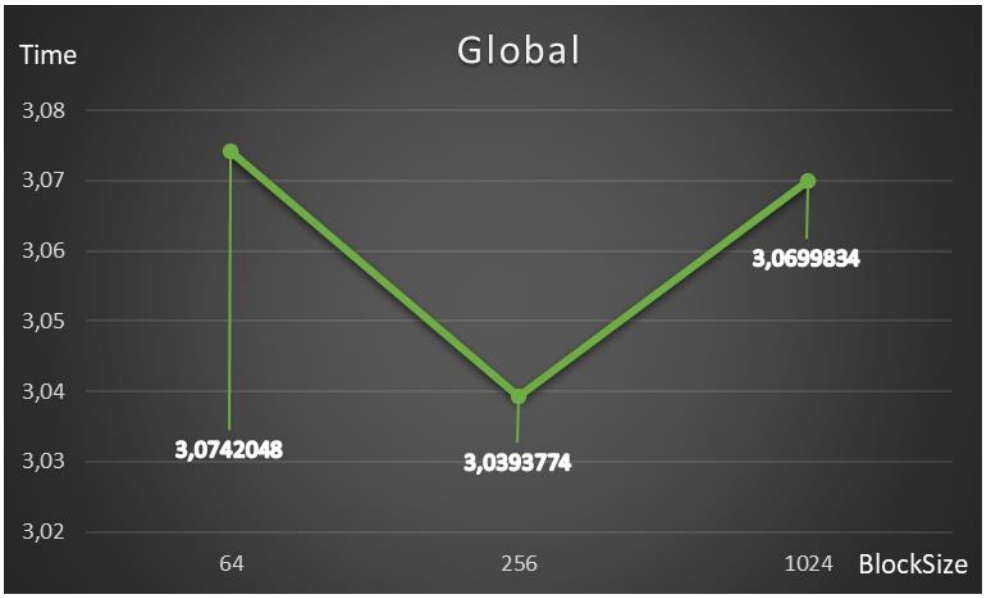
Global Memory

SIZE_1179648-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using global memory. The size of the problem is 1179648 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999	64	18432	0,1702248	3,0742048
1179648	9999	256	4608	0,17432556	3,0393774
1179648	9999	1024	1152	0,17486468	3,0699834



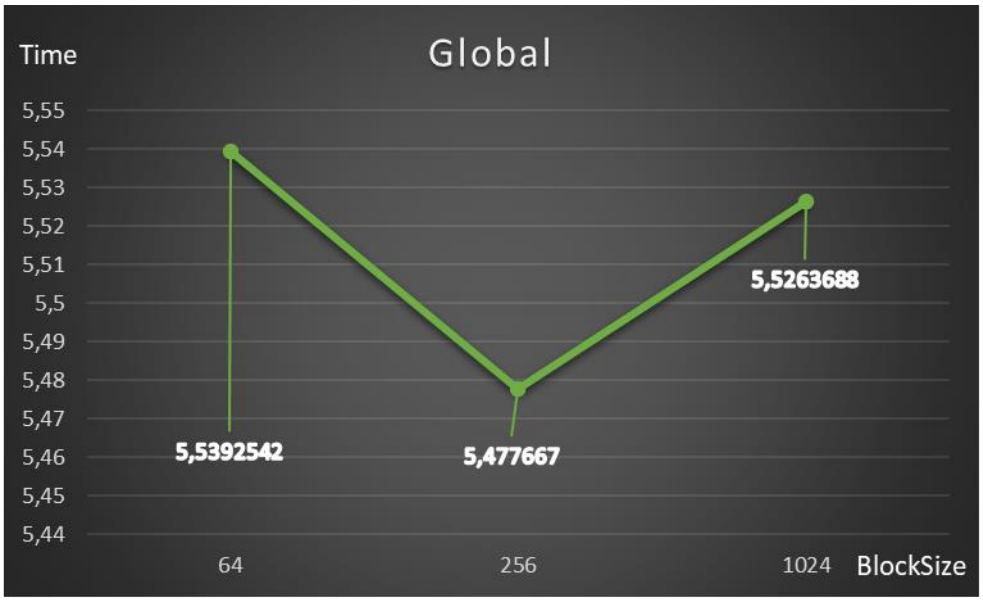
Global-Size1179648-MaxDigit9999

SIZE_1179648-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using global memory. The size of the problem is 1179648 with a maximum number of 9999999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999999	64	18432	0,14653766	5,5392542
1179648	9999999	256	4608	0,1493338	5,477667
1179648	9999999	1024	1152	0,0149271	5,5263688



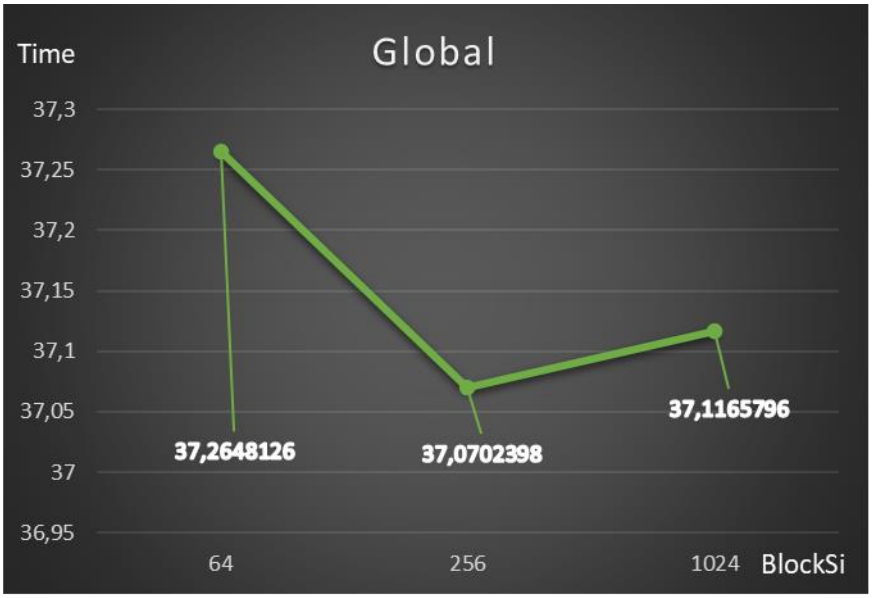
Global-Size1179648-MaxDigit9999999

SIZE_14155776-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using global memory. The size of the problem is 14155776 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999	64	221184	0,1685093	37,2648126
14155776	9999	256	55296	0,17151428	37,0702398
14155776	9999	1024	13824	0,17354666	37,1165796



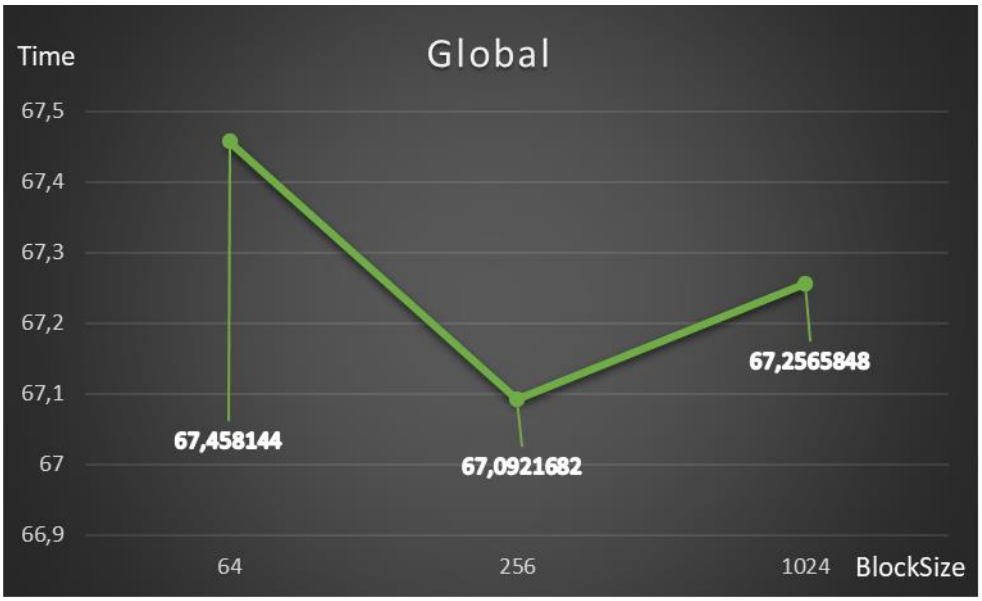
Global-Size14155776-MaxDigit9999

SIZE_14155776-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using global memory. The size of the problem is 14155776 with a maximum number of 9999999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999999	64	221184	0,09308652	67,458144
14155776	9999999	256	55296	0,09476638	67,0921682
14155776	9999999	1024	13824	0,09577396	67,2565848



Global-Size14155776-MaxDigit9999999

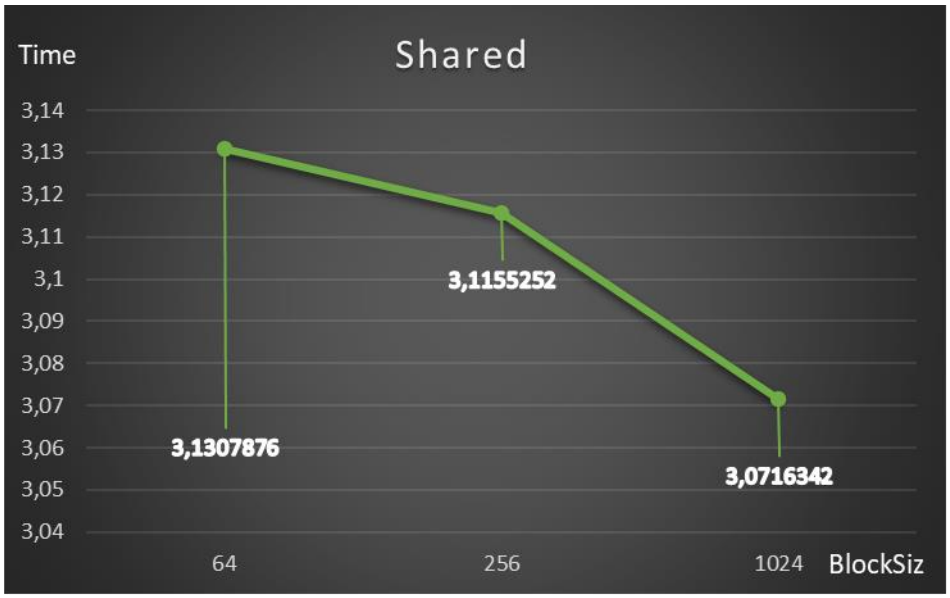
Shared Memory

SIZE_1179648-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using shared memory. The size of the problem is 1179648 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999	64	18432	0,1554352	3,1307876
1179648	9999	256	4608	0,15809694	3,1155252
1179648	9999	1024	1152	0,16258122	3,0716342



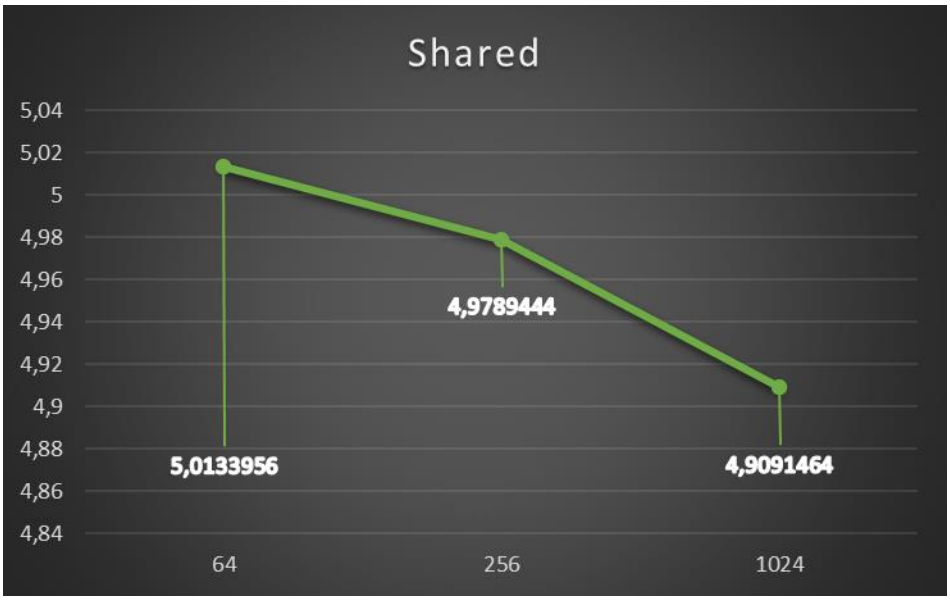
Shared-Size1179648-MaxDigit9999

SIZE_1179648-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using shared memory. The size of the problem is 1179648 with a maximum number of 9999999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999999	64	18432	0,15088302	5,0133956
1179648	9999999	256	4608	0,15330648	4,9789444
1179648	9999999	1024	1152	0,15684476	4,9091464



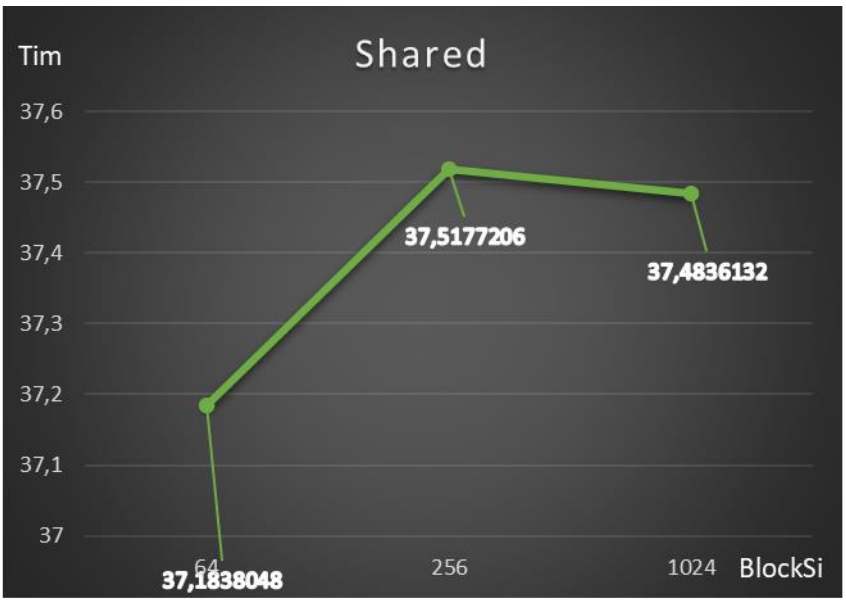
Shared-Size1179648-MaxDigit9999999

SIZE_14155776-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using shared memory. The size of the problem is 14155776 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999	64	221184	0,1570398	37,1838048
14155776	9999	256	55296	0,15754362	37,5177206
14155776	9999	1024	13824	0,1598625	37,4836132



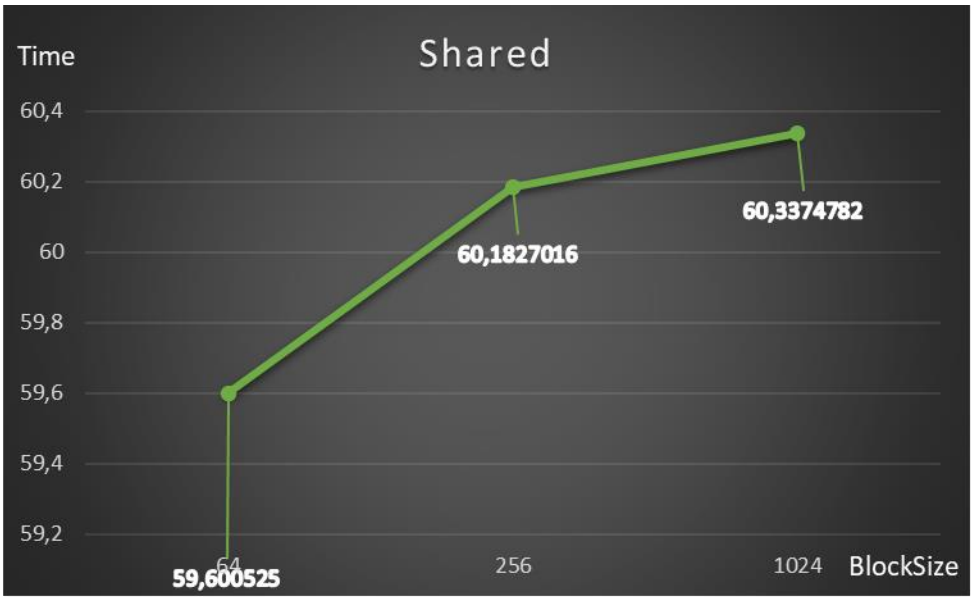
Shared-Size14155776-MaxDigit9999

SIZE_14155776-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using shared memory. The size of the problem is 14155776 with a maximum number of 9999999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999	64	221184	0,15261618	59,600525
14155776	9999	256	55296	0,15219744	60,1827016
14155776	9999	1024	13824	0,15312592	60,3374782



Shared-Size14155776-MaxDigit9999999

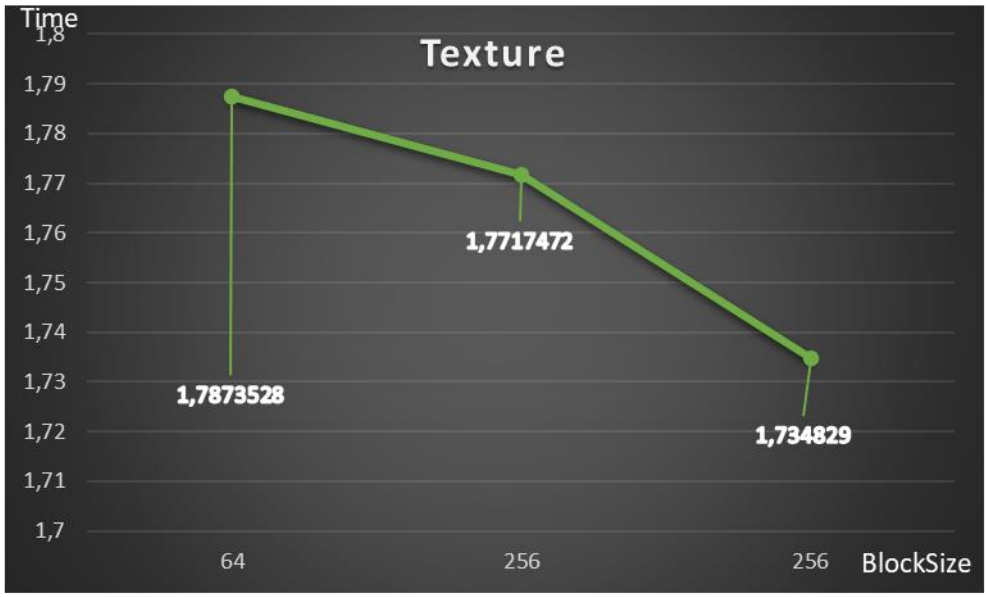
Texture Memory

SIZE_1179648-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using texture memory. The size of the problem is 1179648 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999	64	18432	0,25578848	1,7873528
1179648	9999	256	4608	0,26135894	1,7717472
1179648	9999	1024	1152	0,27086434	1,7348290



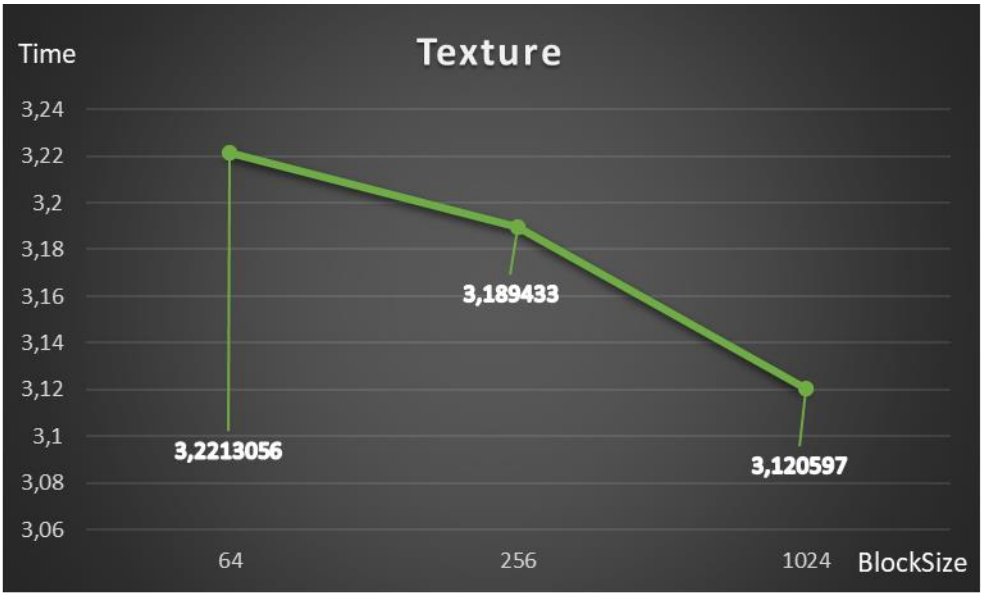
Texture-Size1179648-MaxDigit9999

SIZE_1179648-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using texture memory. The size of the problem is 1179648 with a maximum number of 9999999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999999	64	18432	0,24633262	3,2213056
1179648	9999999	256	4608	0,25039210	3,1894330
1179648	9999999	1024	1152	0,25803984	3,1205970



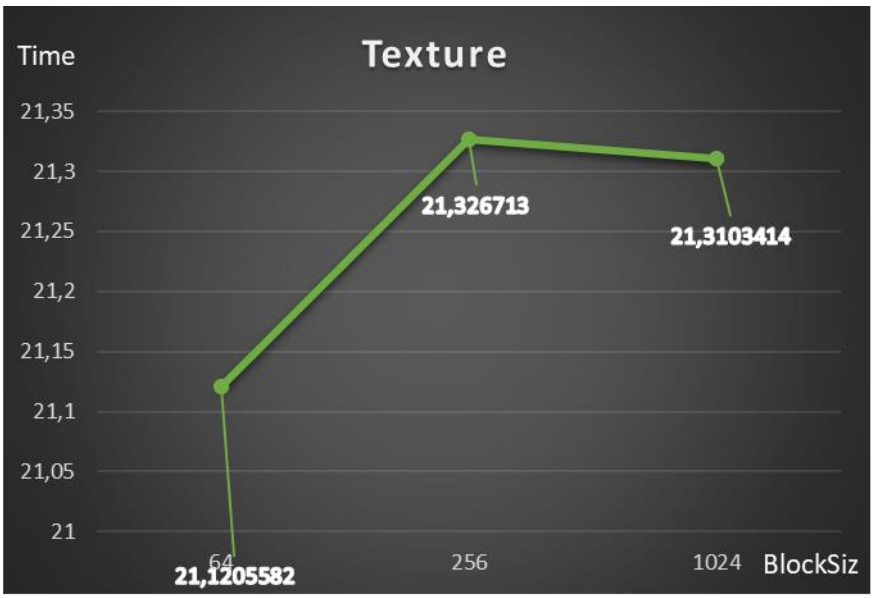
Texture-Size1179648-MaxDigit9999999

SIZE_14155776-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using texture memory. The size of the problem is 14155776 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks. The occupancy is 100%.
- 256: with this choice we have that 2048/256 = 8 blocks. The occupancy is 100%.
- 1024: with this choice we have that 2048/1024 = 2 blocks. The occupancy is 100%.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999	64	221184	0,25972318	21,1205582
14155776	9999	256	55296	0,26055386	21,3267130
14155776	9999	1024	13824	0,26457986	21,3103414



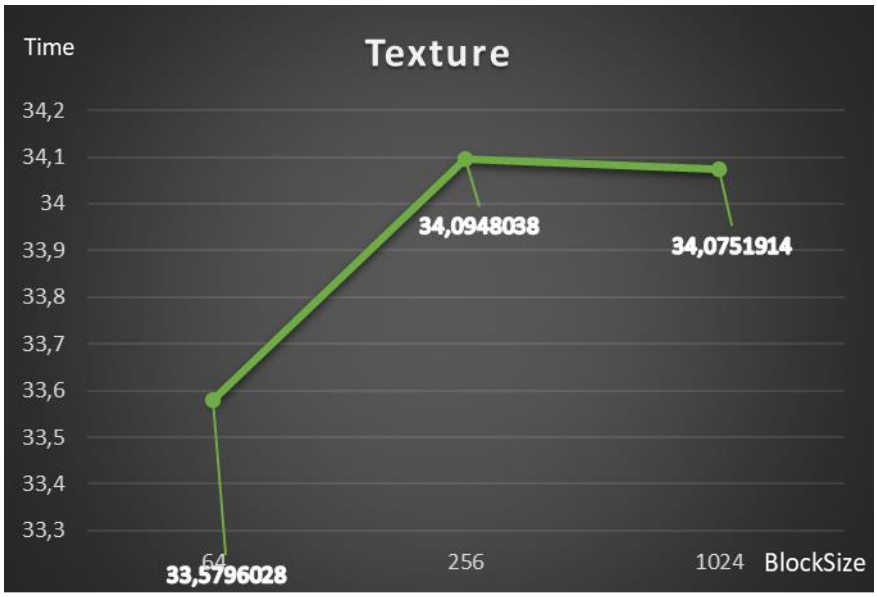
Texture-Size14155776-MaxDigit9999

SIZE_14155776-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using texture memory. The size of the problem is 14155776 with a maximum number of 9999999 and then we evaluate the performances with block size of:

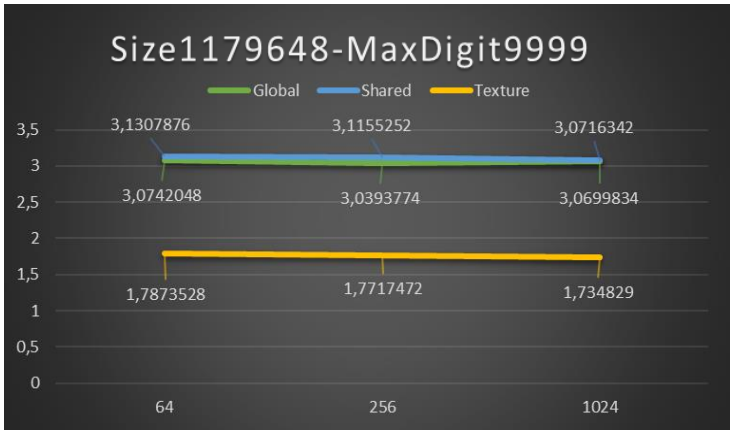
- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

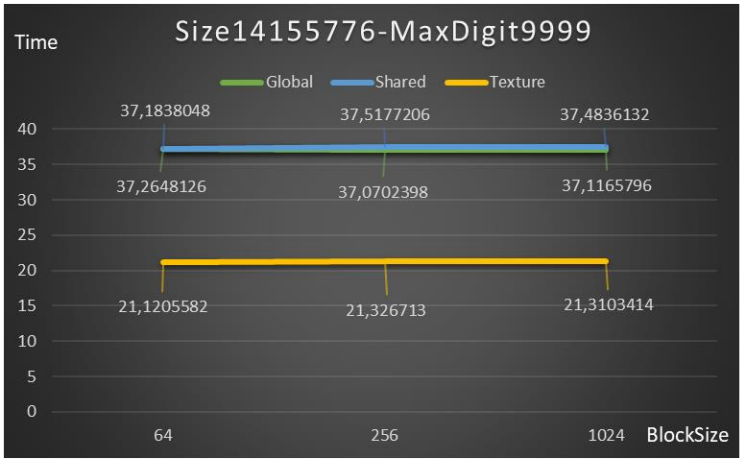
SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999999	64	221184	0,01361278	33,5796028
14155776	9999999	256	55296	0,01358158	34,0948038
14155776	9999999	1024	13824	0,01379002	34,0751914

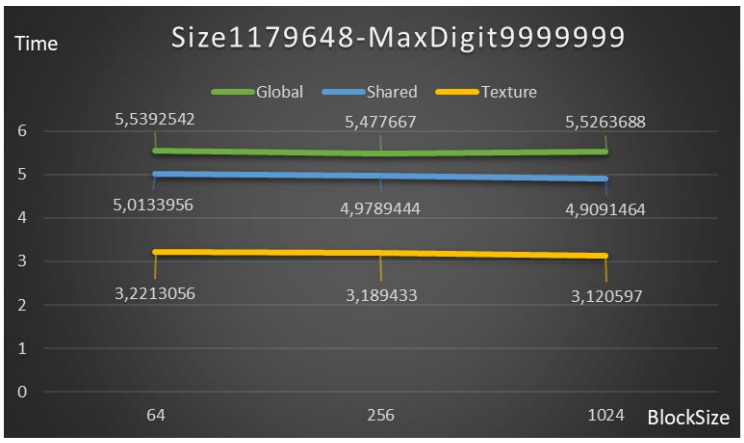


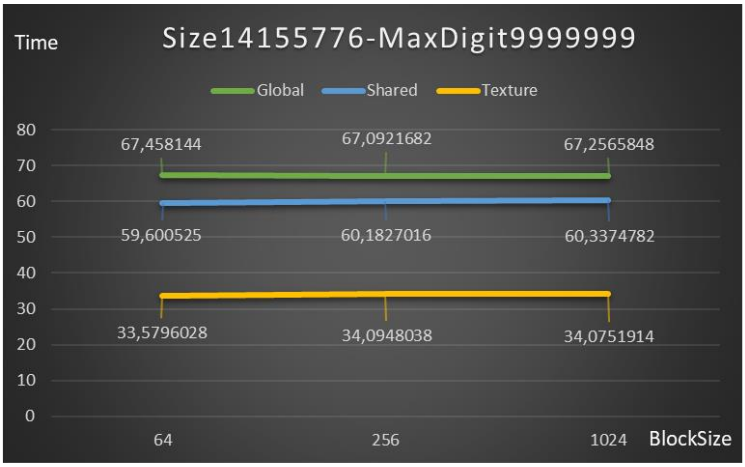
Texture-Size14155776-MaxDigit9999999

Comparison between different type of memory (Global, Shared, Texture)









In these graphs we can perceive the advantage that quick access memories such as Shared and Texture can bring, in fact, as numbers and figures grow, the gap between Global memory, Shared memory, and Shared memory plus Texture memory is evident. This gain is obtained right on the kernel not suitable for execution on the GPU. In fact, this kernel requires that few threads work and so it remains extremely inconvenient for an implementation on GPU, but however it computation requires to access the same elements many times and therefore supports such as the Shared memory and in particular the Texture memory are decisive, whatever the values of "THREADSIZE" and "BLOCKSIZE" are.

Case n° 2 – With CUDA Streams

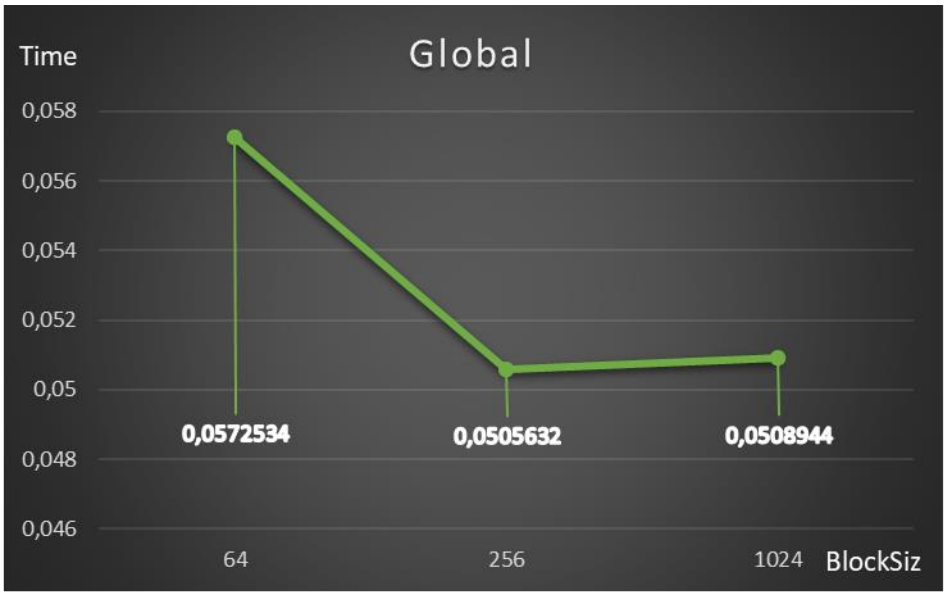
Global Memory

SIZE_1179648-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using global memory and CUDA streams. The size of the problem is 1179648 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999	64	18432	6,34985886	0,0572534
1179648	9999	256	4608	7,26146844	0,0505632
1179648	9999	1024	1152	6,70709508	0,0508944



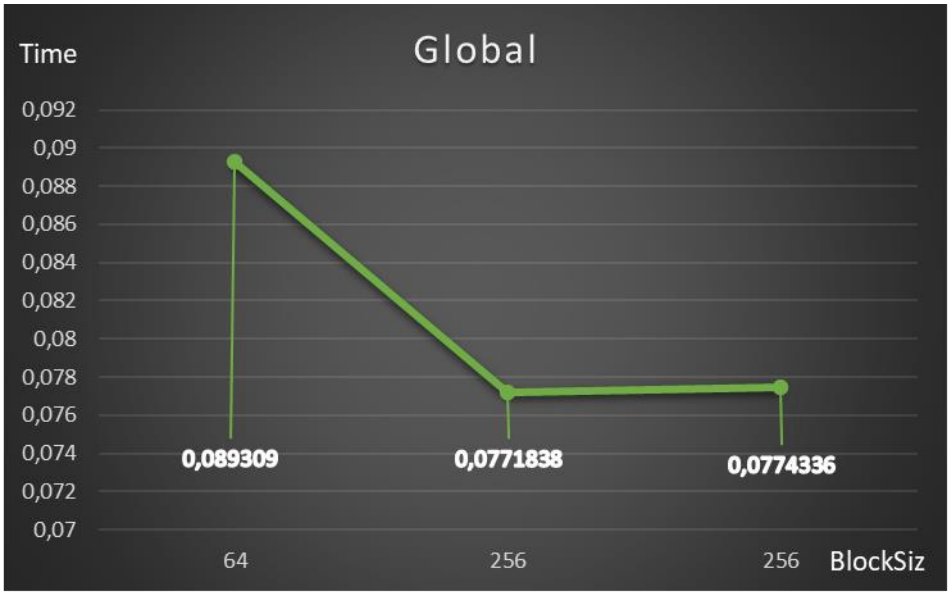
Global-Size1179648-MaxDigit9999

SIZE_1179648-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using global memory and CUDA streams. The size of the problem is 1179648 with a maximum number of 9999999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAX_DIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999999	64	18432	6,22057122	0,0893090
1179648	9999999	256	4608	7,21426596	0,0771838
1179648	9999999	1024	1152	6,59411670	0,0774336



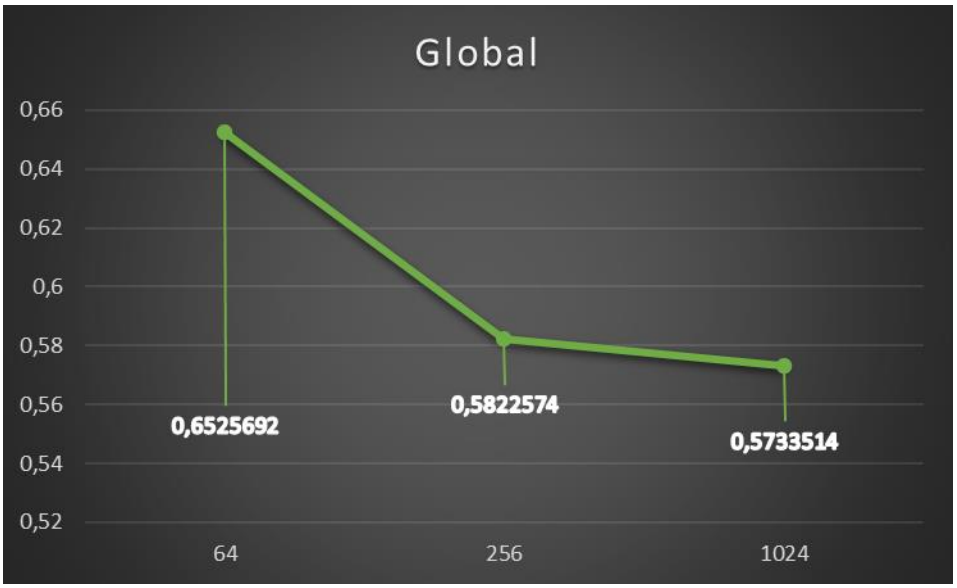
Global-Size1179648-MaxDigit9999999

SIZE_14155776-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using global memory and CUDA streams. The size of the problem is 14155776 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999	64	221184	6,12161986	0,6525692
14155776	9999	256	55296	7,61409222	0,5822574
14155776	9999	1024	13824	7,79136268	0,5733514



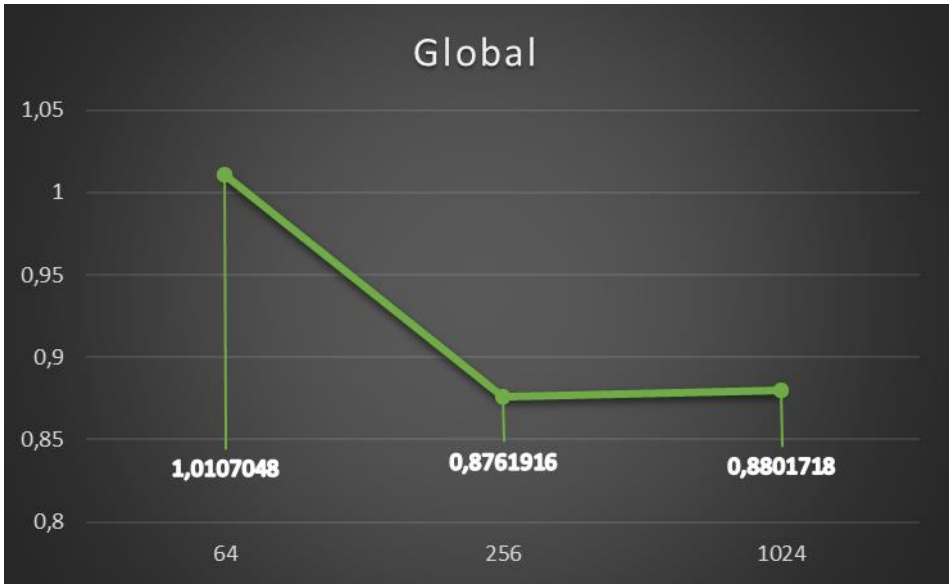
Global-Size14155776-MaxDigit9999

SIZE_14155776-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using global memory and CUDA streams. The size of the problem is 14155776 with a maximum number of 9999999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999999	64	221184	6,0161949	1,0107048
14155776	9999999	256	55296	7,61456502	0,8761916
14155776	9999999	1024	13824	7,65649464	0,8801718



Global-Size14155776-MaxDigit9999999

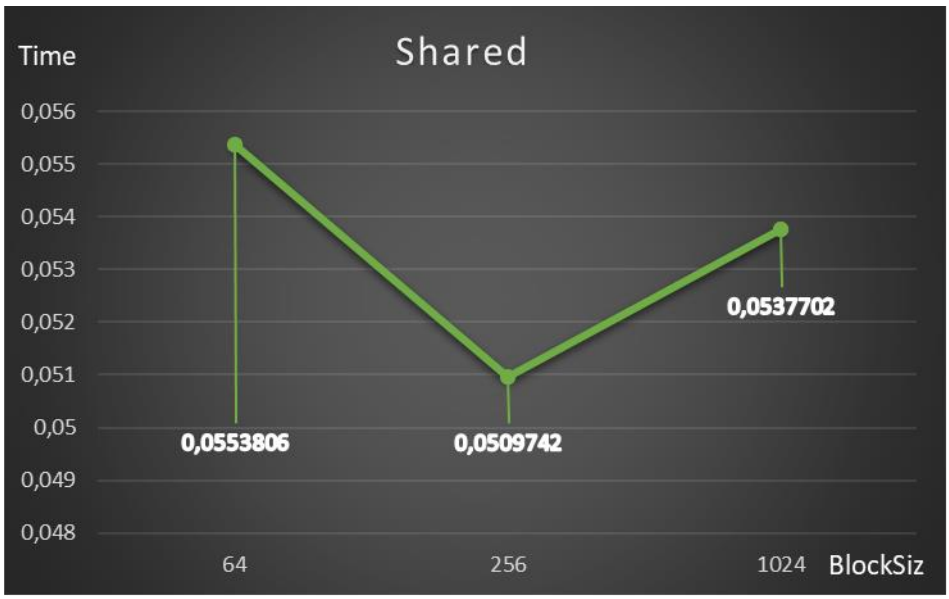
Shared Memory

SIZE_1179648-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using shared memory and CUDA streams. The size of the problem is 1179648 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999	64	18432	5,27132226	0,0553806
1179648	9999	256	4608	5,76289096	0,0509742
1179648	9999	1024	1152	5,57756366	0,0537702



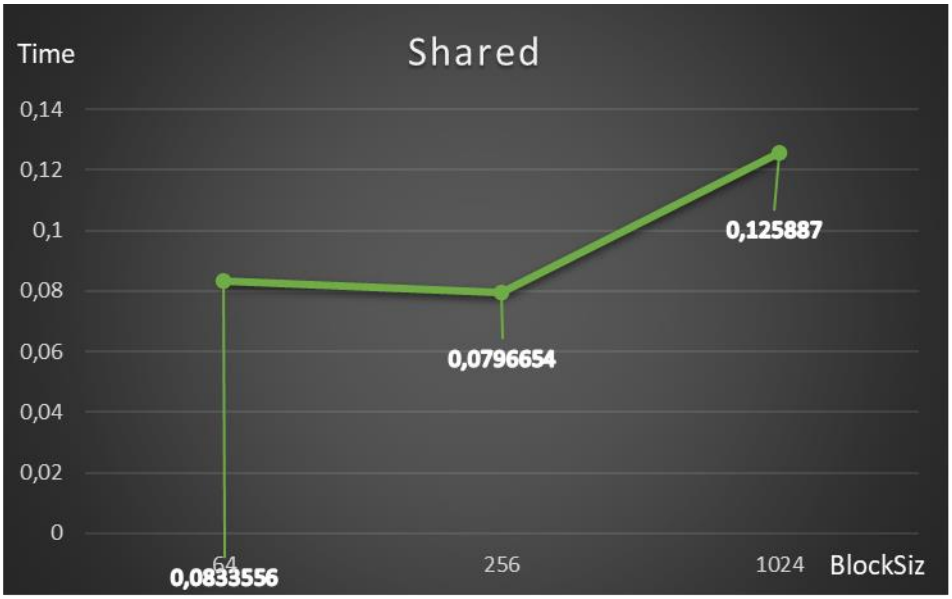
Shared-Size1179648-MaxDigit9999

SIZE_1179648-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using shared memory and CUDA streams. The size of the problem is 1179648 with a maximum number of 9999999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999999	64	18432	5,36182938	0,0833556
1179648	9999999	256	4608	5,57356192	0,0796654
1179648	9999999	1024	1152	5,45982812	0,1258870



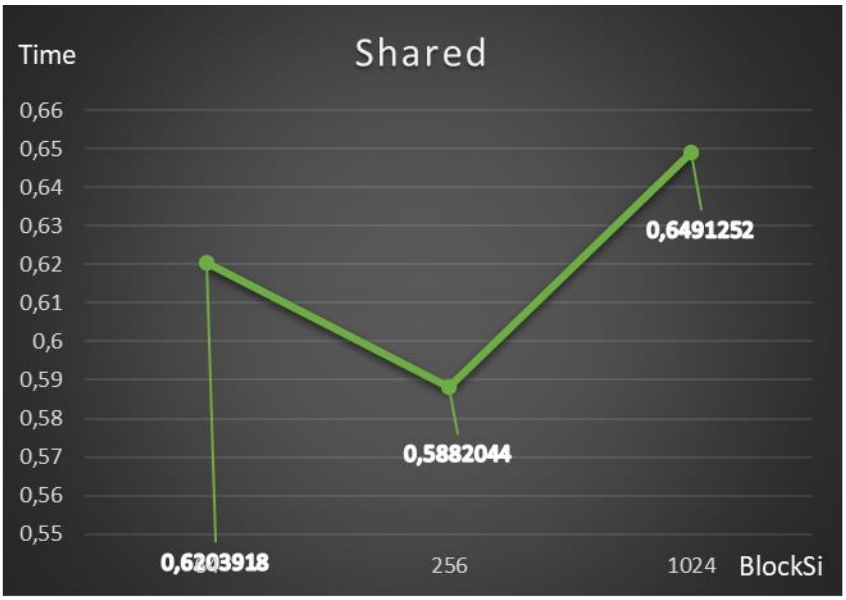
Shared-Size1179648-MaxDigit9999999

SIZE_14155776-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using shared memory and CUDA streams. The size of the problem is 14155776 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999	64	221184	5,64132438	0,6203918
14155776	9999	256	55296	5,99600974	0,5882044
14155776	9999	1024	13824	5,75459420	0,6491252



Shared-Size14155776-MaxDigit9999

SIZE_14155776-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using shared memory and CUDA streams. The size of the problem is 14155776 with a maximum number of 9999999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999999	64	221184	5,58011736	0,955307
14155776	9999999	256	55296	5,98381358	0,897095
14155776	9999999	1024	13824	6,36856698	0,938212

Shared-Size14155776-MaxDigit9999999

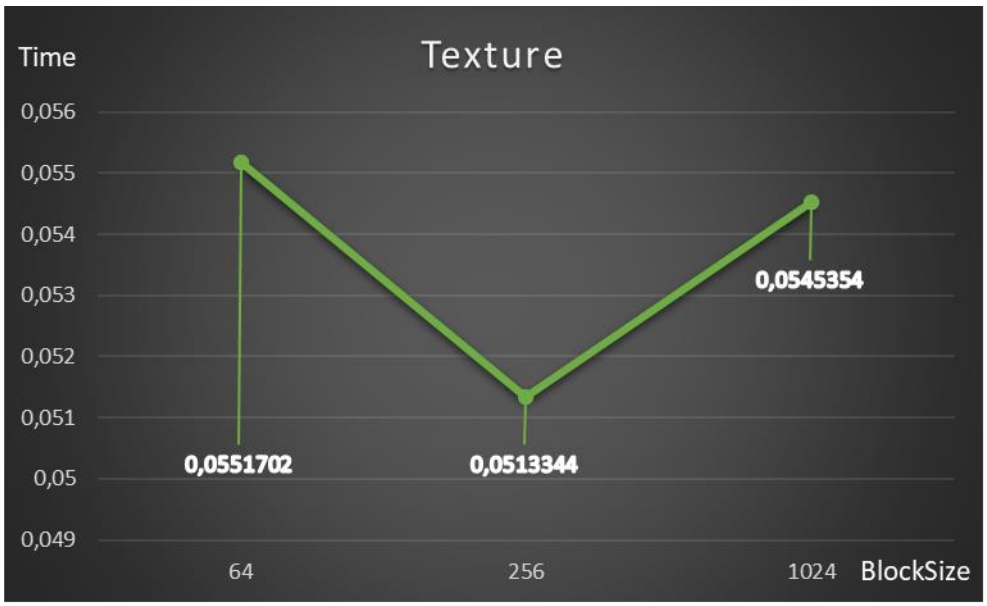
Texture Memory

SIZE_1179648-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using texture memory and CUDA streams. The size of the problem is 1179648 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999	64	18432	5,19006822	0,0551702
1179648	9999	256	4608	5,61607936	0,0513344
1179648	9999	1024	1152	5,38945572	0,0545354



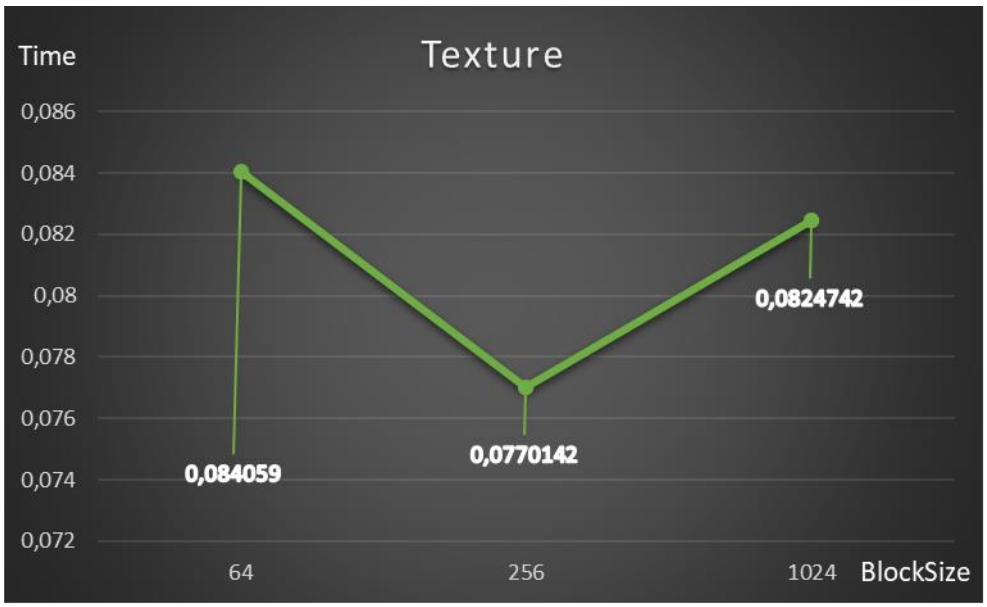
Texture-Size1179648-MaxDigit9999

SIZE_1179648-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using texture memory and CUDA streams. The size of the problem is 1179648 with a maximum number of 9999999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
1179648	9999999	64	18432	5,20016866	0,0840590
1179648	9999999	256	4608	5,65561202	0,0770142
1179648	9999999	1024	1152	5,34661412	0,0824742



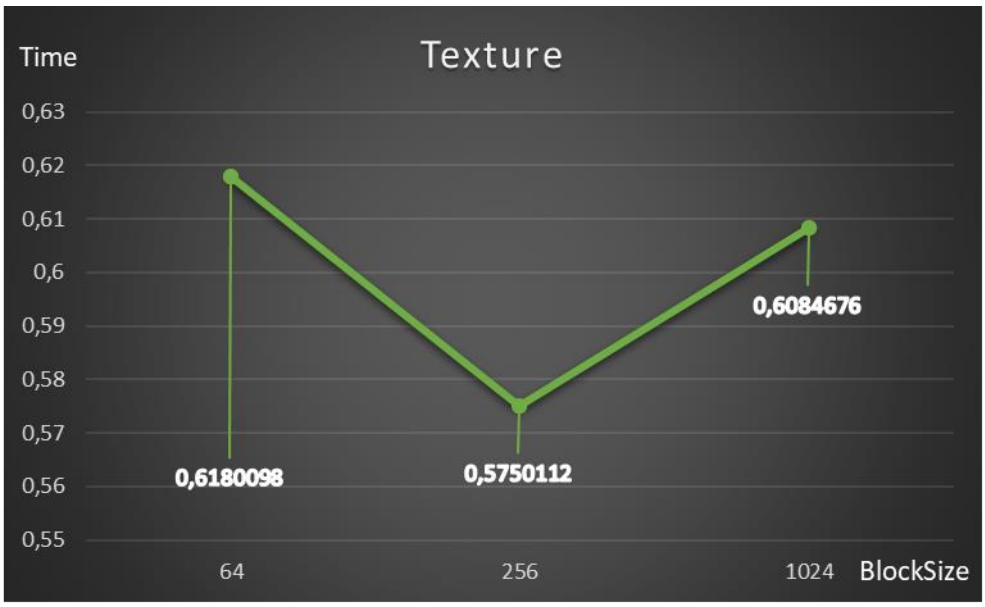
Texture-Size1179648-MaxDigit9999999

SIZE_14155776-MAX_9999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using texture memory and CUDA streams. The size of the problem is 14155776 with a maximum number of 9999 and then we evaluate the performances with block size of:

- 64: with this choice we have that 2048/64 = 32 blocks.
- 256: with this choice we have that 2048/256 = 8 blocks.
- 1024: with this choice we have that 2048/1024 = 2 blocks.

SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999	64	221184	5,52465890	0,6180098
14155776	9999	256	55296	6,00026358	0,5750112
14155776	9999	1024	13824	5,78511184	0,6084676



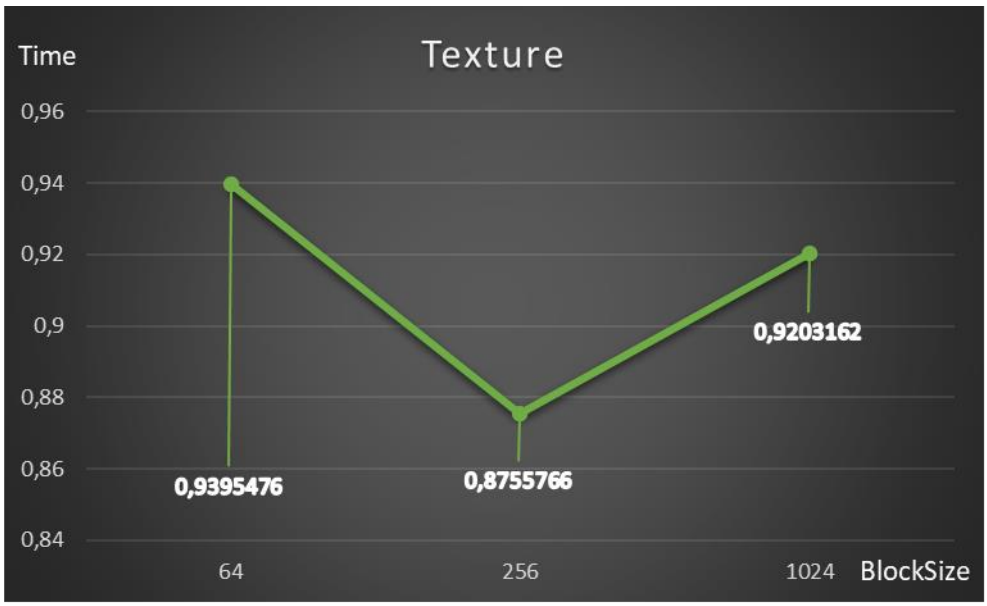
Texture-Size14155776-MaxDigit9999

SIZE_14155776-MAX_9999999

In this case study, we have analysed the problem of parallelizing Radix Sort algorithm using texture memory and CUDA streams. The size of the problem is 14155776 with a maximum number of 9999999 and then we evaluate the performances with block size of:

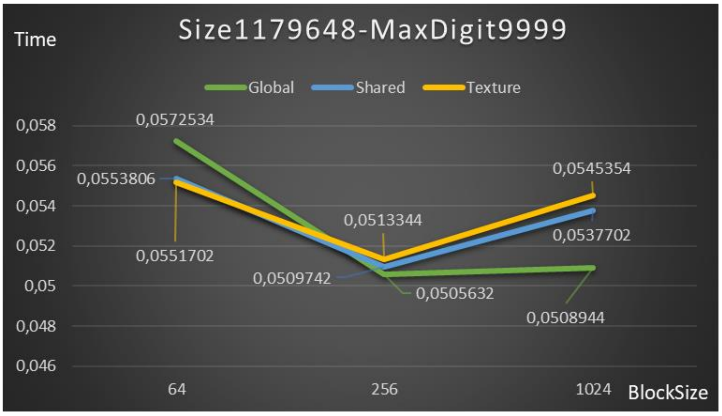
- 64: with this choice we have that 2048/64 = 32 blocks. The occupancy is 100%.
- 256: with this choice we have that 2048/256 = 8 blocks. The occupancy is 100%.
- 1024: with this choice we have that 2048/1024 = 2 blocks. The occupancy is 100%.

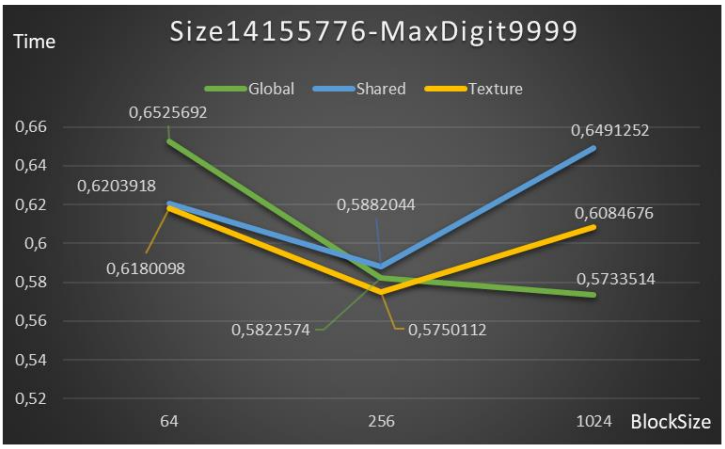
SIZE	MAXDIGIT	BLOCKSIZE	GRIDSIZE	GIPS	TIME_SEC
14155776	9999999	64	221184	5,55191532	0,9395476
14155776	9999999	256	55296	5,96469532	0,8755766
14155776	9999999	1024	13824	5,74021138	0,9203162

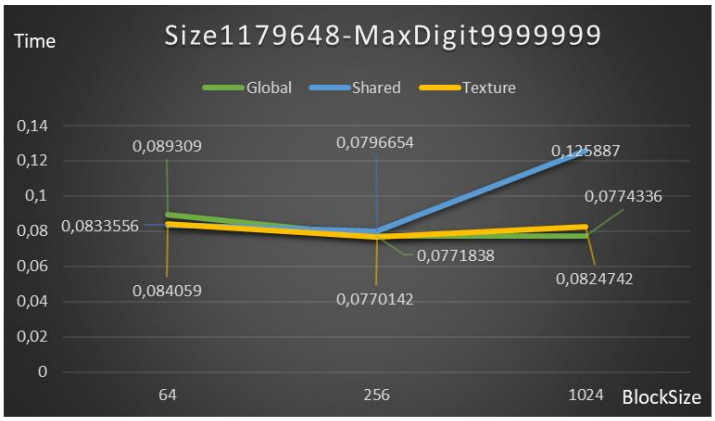


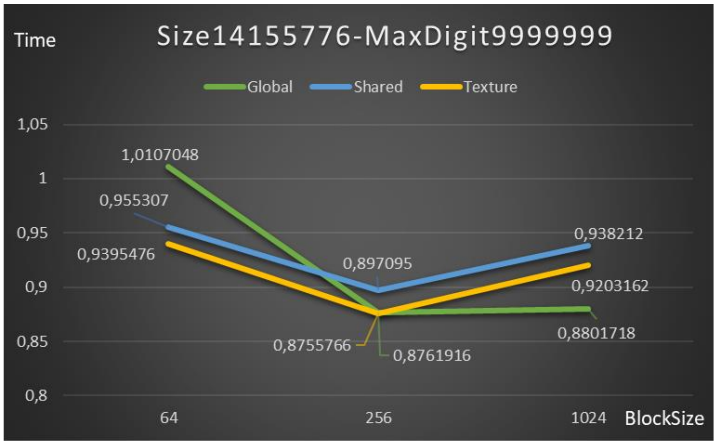
Texture-Size14155776-MaxDigit9999999

Comparison between different type of memory (Global, Shared, Texture)









The introduction of parallel streams and the elimination of the very expensive kernel have caused a flattening of the time required for the execution. However, some considerations can be made: surely it can be perceived that the best configuration in most cases has been the one that uses the Shared memory as the number of digits and the length of the array to be sorted increase. But you can see how even the implementation with the simple use of the Global memory is very fast, often slightly slower than the implementation via Shared memory, both because there are no too expensive accesses to the Global memory, and because there are few independent "accesses" "between shared and global. Last consideration to be made is that as expected, the implementation via Texture memory not only did not bring advantages (since in very rare cases the same location is accessed twice), but rather it was only an additional burden of steps to get to the value contained in the global memory.

Note

As it is shown in the attached Colab Notebook, for the calculation of the GIPS value (Giga IPS), we used the command *nvprof* with the metric *inst_integer* that returns the number of integer instructions for each kernel. We have elaborated these values, converted them in Giga and then we have passed them as input value before running our program for each configuration. So, at the end, as result we obtained the GIPS.

Final Consideration

We decided to test two types of algorithms to solve the sorting through radix sort, both using the counting sort as a base, but what changes are the choices made for the implementation. The first choice concerns the execution of a kernel which in one case is avoided by having those calculations performed on the CPU (much more adequate in that case), in the other the aforementioned kernel is run on the GPU making a very high delay (as extremely disadvantageous for the architecture of the GPU itself).
In fact, this Kernel had a computational structure opposite to the recommended for CUDA Kernels. The kernels must be computationally simple and must be executed by many threads. Instead, the kernel in question (*indexArrayKernel*), was executed by few threads and the required operations were not elementary, so it represented a general slowdown for the entire program.
The second choice, on the other hand, was on further parallelization through multiple CUDA streams; in the best configuration we have, in fact, introduced a further parallelization where possible through streams that have further increased performance by reducing the execution time of the algorithm.

Best configuration of both type of algorithms

As we expected, that's also visible from the graphs, the less efficient configuration of the algorithm has its best implementation in the one that uses both shared memory and texture memory. This is evident from the fact that the already mentioned kernel, that was extremely inefficient for the GPU architecture, gets a high time-boost for the introduction of texture memory.
In fact, by accessing the same memory locations very often through the texture, access times are enormously reduced compared to using only the global memory, almost halving the execution time of the algorithm. Obviously, the implementation on texture also uses shared, so the algorithm also get further advantages due to the speed of the shared.
Looking instead at the best configuration of the algorithm, the best execution times are obtained in the implementation via texture memory, but the improvement obtained compared to that of the shared memory is not that clear as in the previous case of study.This is because the improvements obtained in the previous analysis with texture memory, are lost because the kernel not suitable for the GPU is no longer run, which however, was extremely advantageous solved through texture memory.

Conclusions

From this analysis it can be seen how the use of faster memories such as shared and textures greatly reduce the execution time of the algorithm and therefore are fundamental for an adequate optimization through CUDA. It is also possible to understand how much it is important to exploit them adequately since if misused (therefore not used by exploiting their properties) can result in a simple delay in the execution of the algorithm itself. Furthermore, it is also perceived as an incorrect use of the GPU (even just for a kernel) can enormously deteriorate the performance, making useless the enormous support that an architecture like the GPU's one could provide.

Test case

Testing was done by checking that the resultant array was ordered so, all the numbers in “i” position are greater or equals than “i-1”.
This test wasn’t considered in the “time” of execution.
To test the program is necessary to give as input a value equal to 1 to the TEST define (it is necessary to run the correspondent section in Colab Notebook).
In this way all the execution will be tested and in case of problem an error message is shown, otherwise a success message is shown.
So, we are certain that the algorithm work without problem.
We’re operating with integer values, so we don’t need appropriate functions to compare the numbers.

How to run

Use the colab notebook in the shared directory