Università degli Studi di Salerno
DIEM, a.a 2021-22
Course: HIGH PERFORMANCE COMPUTING
Lecturer: Francesco Moscato

PARALLELIZATION AND PERFORMANCES EVALUATION OF ***COUNTING SORT*** ALGORITHM WITH **CUDA**

Salvatore Grimaldi	0622701742	s.grimaldi29@studenti.unisa.it
Enrico Maria Di Mauro	0622701706	e.dimauro5@studenti.unisa.it
Allegra Cuzzocrea	0622701707	a.cuzzocrea2@studenti.unisa.it

Index

Introduction

The main purpose of this report is parallelizing and evaluating performances of Counting Sort Algorithm. CUDA is the parallel computing platform and API used to obtain the results.

In the following pages the problem faced is going to be described in a detailed way, along with the most important theoretical concepts about GPUs (Graphics Processing Unit). Great attention is going to be reserved to the description of the different case studies considered: the results are going to be analysed and explained in the light of the theoretical knowledge acquired during the High-Performance Computing course held by prof. Francesco Moscato at University of Salerno.

Problem description

The problem is how to parallelize and evaluate performances of **Counting Sort Algorithm**, by using **CUDA**. Counting sort is an integer sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array that is then used to get the actual sorted array.

Counting sort can be used only to sort collections of objects whose keys are positive integers: this is the reason why it is described as an integer sorting algorithm. Counting sort is not a comparison sort, which means that it is not based on comparisons among the objects of the collection meant to be sorted. Counting sort running time is linear in the number of items and the difference between the maximum key value and the minimum key value, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items.

It is important to observe that in the following case studies the minimum key value is assumed 0, while the maximum key value is given as input to the program, so it won't be necessary for the counting sort function finding the minimum and the maximum in the collection. Moreover, to make things easier and faster to understand, it has been chosen to deal with a collection that is a simple array of integers.

Counting sort algorithm can be divided in few steps:

1. First, an auxiliary array **c** must be allocated. Its size is equal to $maximum - minimum + 1$, which means that in our version its size is equal to $maximum + 1$, where maximum, as said before, is an input. Furthermore, for notation simplicity, the array meant to be sorted is called **a**.
2. **c** must be initialized to 0.
3. A for-loop traverses **a** and at each iteration increments by 1 the **c** element placed in the position corresponding to the visited **a** element.
4. A for-loop increments every **c** element by adding to it the sum of all its previous elements in **c**.
5. A for-loop exploits **a** and **c** to build the sorted array **b**.
6. A for-loop copies **b** in **a**.

CUDA is the parallel computing platform and API used to parallelize counting sort algorithm in this report. It is extremely powerful because allows programmers to use certain types of graphics processing unit (**GPU**) for general purpose processing: an approach called General-Purpose computing on GPUs (**GPGPU**).

CUDA can be seen as a software layer that gives direct access to the GPU’s virtual instruction set and parallel computational elements, for the execution of compute kernels. This computing platform is designed to work with programming languages such as C, C++, and Fortran.

In this report we deal with C language.

Programming in CUDA requires at least basic knowledge of a GPU architecture (how threads work in parallel, how the memory is organized, etc.). In the next few pages, we are going to observe the importance of this knowledge to get faster and more efficient programs.

Before starting the discussion about case studies and the results obtained, it is relevant to say that the environment used to write code, execute programs and make measures is **Colab**.

Theoretical notes

GPU stands for **Graphics Processing Unit** and is a specialized electronic circuit designed to rapidly manipulate and alter memory to accelerate the creation of images in a frame buffer intended for output to a display device.

GPUs are used in embedded systems, mobile phones, personal computers, workstations, and game consoles.

Since the beginning of the century GPUs have been used not only for graphic computations but also for general-purpose computing (typically done by CPUs).

The main differences between GPUs and CPUs are:

1. GPUs have much more parallel execution units than CPUs.
2. GPUs have deeper pipes than CPUs.
3. CPUs can handle complex control logic, while GPUs are optimized for simple control logic.

GPUs are perfect to run in parallel the same set of simple instructions on a lot of different data.

To fully understand the results of this report, it is fundamental to give some definitions:

- **Host** is the computer on which GPU is mounted.
- **Device** is another word through which GPU is indicated.
- **Compute Capability** (c.c.) identifies the features supported by the GPU hardware (its version).
- **Kernel** is a CUDA function written to run on device.
- **Execution cores** are the computing elements mounted on a GPU. They are a sort of ALUs (Arithmetic Logic Unit).
- **Streaming Multiprocessor** (SM) is a GPU unit that consists of registers, several caches, warp schedulers and execution cores.
- **Thread** is a flow of execution.
- **Block** is a group of threads organized in 1D, 2D or 3D logical arrays.
- **Grid** is a group of blocks organized in 1D, 2D or 3D logical arrays.

GPUs are characterized by a particular memory hierarchy.

All CUDA threads in a block have access to:

- resources of the SM assigned to the block to which the threads belong:
  - **Registers**
  - **Shared Memory**

  Threads belonging to different blocks cannot share registers and shared memory.

- all the memory types available on the GPU are:
  - **Global Memory**
  - **Constant Memory** (read only)
  - **Texture Memory** (read only)

CPU can access and initialize both constant and texture memories.

Global, constant and texture memories have persistent storage duration.

To fully understand the different case studies analysed in this report, it is compulsory to know the main differences among global, shared and texture memories.

- **Global Memory** is the largest memory available on a device. It is comparable to a RAM for CPU and its status is maintained among different kernel launches. Furthermore, it can be accessed both read/write from all threads of the kernel grid.
  It is the unique memory that can be used in read/write access from the CPU. It has a very high bandwidth (throughput up to 144-177 GB/s) and a very high latency (about 400-800 clock cycles).
- **Shared Memory** is fast because its buffer is physically placed on GPU. For this reason, it has a very low latency (about 10 clock cycles). Shared memory is about 64 kb/SM and each SM has its own memory shared block. A shared variable must be declared in the kernel so that CUDA compilator creates a variable for each block. Shared memory is quickly accessible by all threads in a block. This type of memory is not persistent, that means its status is not maintained between different kernel calls.
- **Texture Memory** is not placed on chip, but it is cached on it, which means that subsequent uses of the same texture memory are extremely fast. It can be utilized to obtain a greater efficient bandwidth, reducing calls to global memory on DRAM. Cache associated to the texture memory is small (about 64 bK/SM). This kind of memory is read-only and is mostly thought for vector images, which have a meaningful spatial location. Finally, each SM has various texture fetch units that are dedicated units to access the texture memory.

Experimental setup

Information provided below refers to hardware and software used during the evaluation of performances.

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
power management  :

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
power management  :
```

MEM

```
MemTotal:        13302912 kB
MemFree:           582208 kB
MemAvailable:    12341928 kB
Buffers:           302168 kB
Cached:          11151500 kB
SwapCached:             0 kB
Active:           2405352 kB
Inactive:         9598108 kB
Active(anon):      483524 kB
Inactive(anon):       472 kB
Active(file):     1921828 kB
Inactive(file):   9597636 kB
Unevictable:            0 kB
Mlocked:                0 kB
SwapTotal:              0 kB
SwapFree:               0 kB
Dirty:                280 kB
Writeback:              0 kB
AnonPages:         549860 kB
Mapped:            245728 kB
Shmem:               1268 kB
KReclaimable:      566308 kB
Slab:              617388 kB
SReclaimable:      566308 kB
SUnreclaim:         51080 kB
KernelStack:         4992 kB
PageTables:          7872 kB
NFS_Unstable:           0 kB
Bounce:                 0 kB
WritebackTmp:           0 kB
CommitLimit:      6651456 kB
Committed_AS:     3189576 kB
VmallocTotal:34359738367 kB
VmallocUsed:        44944 kB
VmallocChunk:           0 kB
Percpu:              1448 kB
AnonHugePages:          0 kB
ShmemHugePages:         0 kB
ShmemPmdMapped:         0 kB
FileHugePages:          0 kB
FilePmdMapped:          0 kB
CmaTotal:               0 kB
CmaFree:                0 kB
HugePages_Total:        0
HugePages_Free:         0
HugePages_Rsvd:         0
HugePages_Surp:         0
Hugepagesize:        2048 kB
Hugetlb:                0 kB
DirectMap4k:       201536 kB
DirectMap2M:      6086656 kB
DirectMap1G:      9437184 kB
```

DSK

```
Filesystem      Size  Used Avail Use% Mounted on
overlay          79G   43G   37G  54% /
tmpfs            64M     0   64M   0% /dev
shm             5.7G     0  5.7G   0% /dev/shm
/dev/root       2.0G  1.2G  817M  59% /sbin/docker-init
tmpfs           6.4G   36K  6.4G   1% /var/colab
/dev/sda1        86G   47G   40G  55% /opt/bin/.nvidia
tmpfs           6.4G     0  6.4G   0% /proc/acpi
tmpfs           6.4G     0  6.4G   0% /proc/scsi
tmpfs           6.4G     0  6.4G   0% /sys/firmware
drive            79G   44G   35G  57% /content/drive
```

GPU (essential)

```
Device number: 0
  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
```

GPU (complete)

```
Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.2 / 9.2
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes (11996954624 
bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 
3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           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
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 4
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device 
simultaneously) >
```

BANDWIDTH

```
Device 0: Tesla K80
 Range Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   1000                         268.8
   101000                       6211.3
   201000                       6639.9
   301000                       6856.2
   401000                       6912.0
   501000                       7130.6
   601000                       7196.2
   701000                       7309.9
   801000                       7320.7
   901000                       7292.3

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   1000                         411.6
   101000                       6442.9
   201000                       7033.6
   301000                       7240.8
   401000                       7408.8
   501000                       7475.6
   601000                       7526.6
   701000                       7564.5
   801000                       7608.7
   901000                       7614.9

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(MB/s)
   1000                         257.1
   101000                       26773.3
   201000                       44430.2
   301000                       57393.3
   401000                       75088.5
   501000                       83781.6
   601000                       89940.2
   701000                       103833.5
   801000                       91738.3
   901000                       80863.3
```

Software

```
Ubuntu       : 18.04.5 LTS
GCC          : 7.5.0
NVCC (CUDA)  : 9.2.88
NVIDIA-SMI   : 460.32.03
```

Case studies and performances

In this report three different case studies are considered:

- **Case Study n.1**: only global memory is exploited.
- **Case Study n.2**: shared memory and global memory are both used.
- **Case Study n.3**: texture memory, shared memory and global memory are used all together.

In all the case studies the sizes considered are 1mln, 10mln and 100mln, where sizes are the dimensions of the arrays meant to be sorted. Moreover, two ranges are taken in account: 100 and 1000. The range represents the maximum integer that can be found in the unsorted array.

The choice regarding the use of multiple sizes and ranges is determined by the intention of testing the counting sort algorithm launched on GPU in different conditions. One of the purposes is comparing algorithm performances for different sizes and ranges. What it is expected is that: fixing the size, higher ranges imply greater execution time, while fixing the range, greater sizes imply greater execution time.

Furthermore, it is expected that the program exploiting only global memories results slower than the one exploiting both global and shared memories, and the one exploiting shared, texture and global memories is faster than the two previous ones.

Program structure (common to all case studies)

The **main** function requires 3 inputs: **size** (array length), **range** and **block size**, checks them and calls the following functions:

- **initArray()**: this function realizes random unsorted array initialization by calling the kernel gpu_initArray() and measures its execution time.
- **countingSortDEVICE()**: this function applies the counting sort algorithm in order to sort the previously created random array by calling the kernels gpu_fullC(), gpu_sumC(), gpu_lastKernel() and measures the execution time needed for them.
- **countingSortHOST()**: this function applies the sequential version of counting sort algorithm in order to sort a copy of the previously created random array. It is necessary in order to determine whether countingSortDEVICE() produces a correct result or not.
- **make_csv()**: this function creates, if not existing, a custom file .csv which is meant to contain info about block size, grid size and execution times referred to each program execution.

Preliminary considerations

Before introducing the case studies, the information reported in Experimental Setup section can be exploited to establish the ideal number of thread blocks per SM for Tesla K80.

The most important data to consider are:

- max threads per SM: 2048
- max blocks per SM: 16 (this info can be found here)
- max threads per block = max block size = 1024

Hypothesis 1: block size = 32 (threads)
estimated blocks per SM = 2048 / 32 = 64
but 64 > 16 → not good
in fact actual used threads per SM are only: 16 * 32 = 512
512 < 2048 → we are not exploiting all threads per SM
25% occupancy because (512 / 2048) * 100 = 25

Hypothesis 2: block size = 64 (threads)
estimated blocks per SM = 2048 / 64 = 32
but 32 > 16 → not good
in fact actual used threads per SM are only: 16 * 64 = 1024
1024 < 2048 → we are not exploiting all threads per SM
50% occupancy because (1024 / 2048) * 100 = 50

Hypothesis 3: block size = 128 (threads)
estimated blocks per SM = 2048 / 128 = 16
16 = 16 → good
actual used threads per SM are: 16 * 128 = 2048
which means that we have 100% occupancy

Hypothesis 4: block size = 256 (threads)
estimated blocks per SM = 2048 / 256 = 8
8 < 16 → good
actual used threads per SM are: 8 * 256 = 2048
which means that we have 100% occupancy

Hypothesis 5: block size = 512 (threads)
estimated blocks per SM = 2048 / 512 = 4
4 < 16 → good
actual used threads per SM are: 4 * 512 = 2048
which means that we have 100% occupancy

Hypothesis 6: block size = 1024 (threads) THIS IS THE MAX
estimated blocks per SM = 2048 / 1024 = 2
2 < 16 → good
actual used threads per SM are: 2 * 1024 = 2048
which means that we have 100% occupancy

In the light of these measures, it is clear that block sizes equal to 32 and 64 are not ideal because they are not able to exploit all threads resident in a SM.
Block sizes equal to 128, 256, 512, 1024, instead, ensure maximum SM utilization.
In particular, it can be observed that from 128 to 1024, increasing the block size, the number of resident blocks per SM decreases from 16 to 2.

Being aware of this is significant because every SM has a maximum number of 32-bit registers that can be distributed among resident blocks. Specifically, Tesla K80 has a maximum of 128k 32-bit registers per SM. To establish the number of 32-bit registers assigned to each resident block, it is possible to divide the number of available registers on SM by the number of resident blocks. This means that increasing the number of resident blocks, the number of available 32-bit registers for each block decreases.

Another meaningful resource provided by each SM is shared memory. Especially, there is a maximum quantity of shared memory provided by SM, which can be divided among the resident blocks so that each of them has its own independent portion. Specifically, shared memory dimension per SM for Tesla K80 is 128KB. Moreover, another parameter to consider is the maximum dimension of shared memory portion per block, that in Tesla K80 case is 48KB. This means that having only one resident block per SM does not allow to exploit all SM shared memory: to fully exploit it, at least 3 resident blocks (128KB/48KB = 2.3) are needed.

Graphic and table legend

The graphic and table legend useful to read the results is reported below:

- **blockSize**: thread block dimension
- **gridSize**: grid dimension, expressed in number of blocks
- **elapsedInit**: execution time (in seconds) for the unsorted array initialization
- **elapsedSort**: execution time (in seconds) for the counting sort algorithm
- **MIPSSort**: million instructions per second corresponding to the counting sort section of the program

- **Regs**: Number of registers used per CUDA thread. This number includes registers used internally by the CUDA driver and/or tools and can be more than what the compiler shows
- **DSMEM**: Dynamic shared memory allocated per CUDA block
- **Size**: the amount of transferred/set data
- **Throughput**: memory transfer throughput
- **Name**: kernel execution or memory copy/set instance

Case Study n.1 – Global Memory

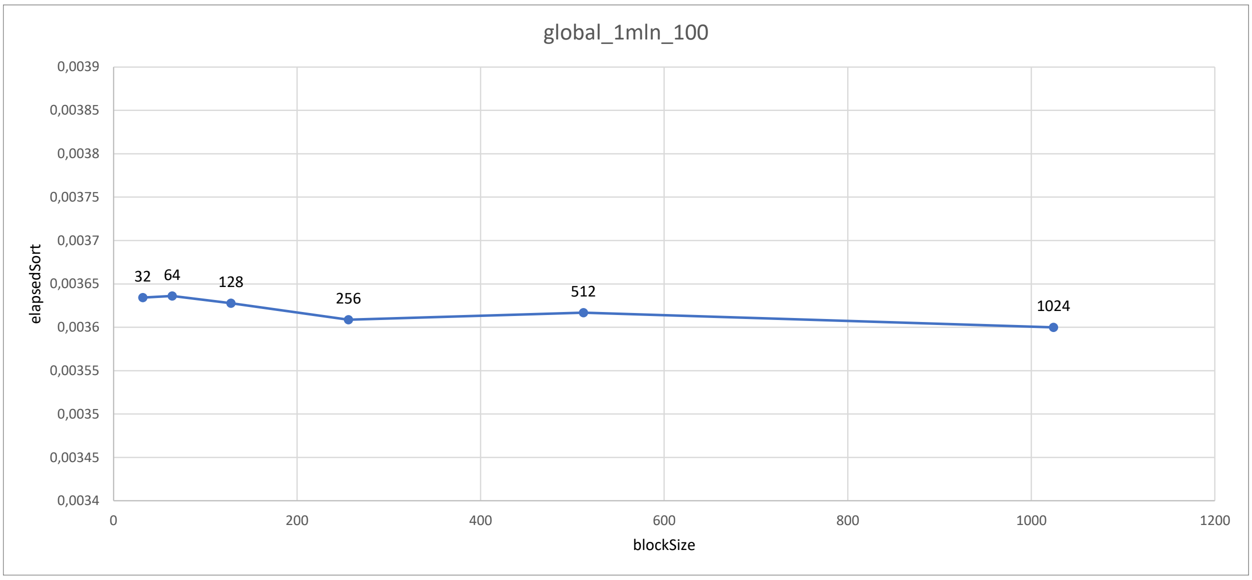
In this case study the only memory exploited among global, shared and texture is the global memory.

The kernels behaviour is as follows:

- **gpu_initArray**: this kernel is the same for all the case studies considered and its logic is very easy. First, thread index is computed thanks to the typical formula $blockIdx.x * blockDim.x + threadIdx.x$, so that each thread can insert in a different position of the unsorted array **a** a random integer. It is interesting to notice that the randomization is obtained through the use of functions *curand_init()* and *curand()*, provided by cuRAND API.
- **gpu_fullC**: this kernel is meant to full the auxiliary array **c** by incrementing by 1 the **c** element placed in the position corresponding to the current visited unsorted array element. Its logic is linear and easy to understand: thread index is computed using the same formula seen above, so that each thread takes care of a particular unsorted array element and increments the right **c** element. It can be noticed that, in order to avoid race conditions among threads, the specific function *atomicAdd()* has been used.
- **gpu_sumC**: this kernel is launched on only one thread because of its particular logic. It actually presents a for-loop that iterates over **c**, incrementing every **c** element by adding to it the sum of all its previous elements in **c**.
- **gpuLastKernel**: this kernel sorts **a** using **c** and puts the result in another array, whose name is **sorted**, passed as parameter. First of all, thread index is computed using always the same formula, so that each thread can read a different **a** element and update as consequence the correct **sorted** position.

SIZE-1mln-RANGE-100

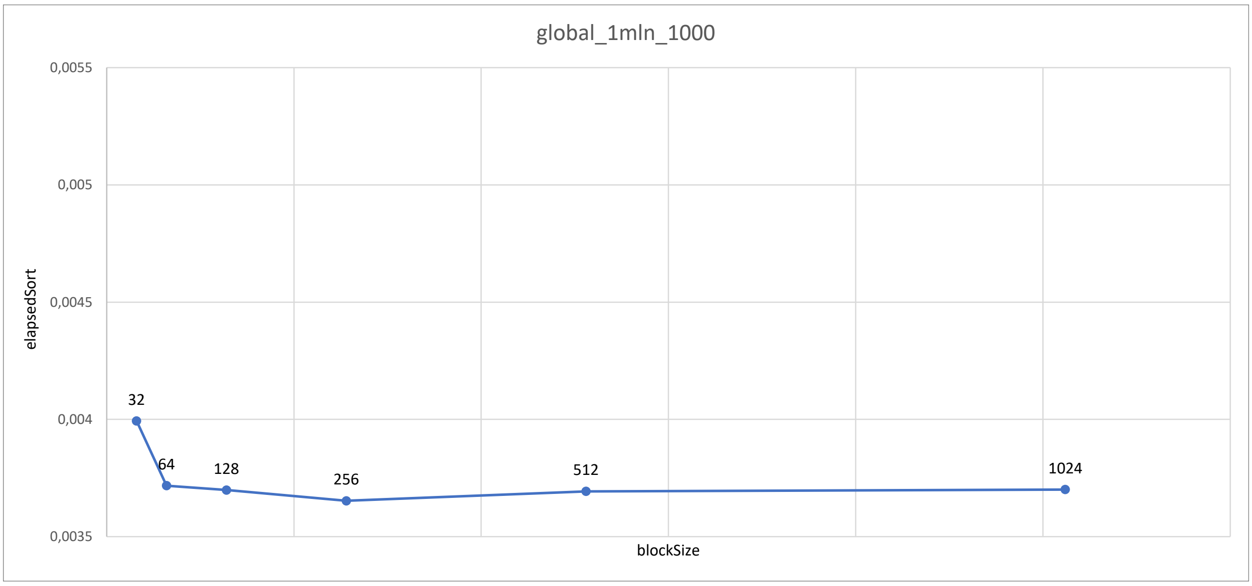
blockSize	gridSize	elapsedSort	MIPSSort
32	31250	0,00363416	4127,61051
64	15625	0,00363604	4125,47634
128	7813	0,00362772	4134,93792
256	3907	0,00360868	4156,75455
512	1954	0,0036168	4147,42231
1024	977	0,00359984	4166,96214



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		3.814697	2.273570	[CUDA memcpy DtoH]
		3.814697	6.686333	[CUDA memcpy HtoD]
		0.000385	0.079446	[CUDA memset]
8	0			gpu_fullC(int*, int*, int)
20	0			gpu_sumC(int*, int)
9	0			gpu_lastKernel(int*, int*, int*, int)
		3.814697	6.711400	[CUDA memcpy DtoH]

SIZE-1mln-RANGE-1000

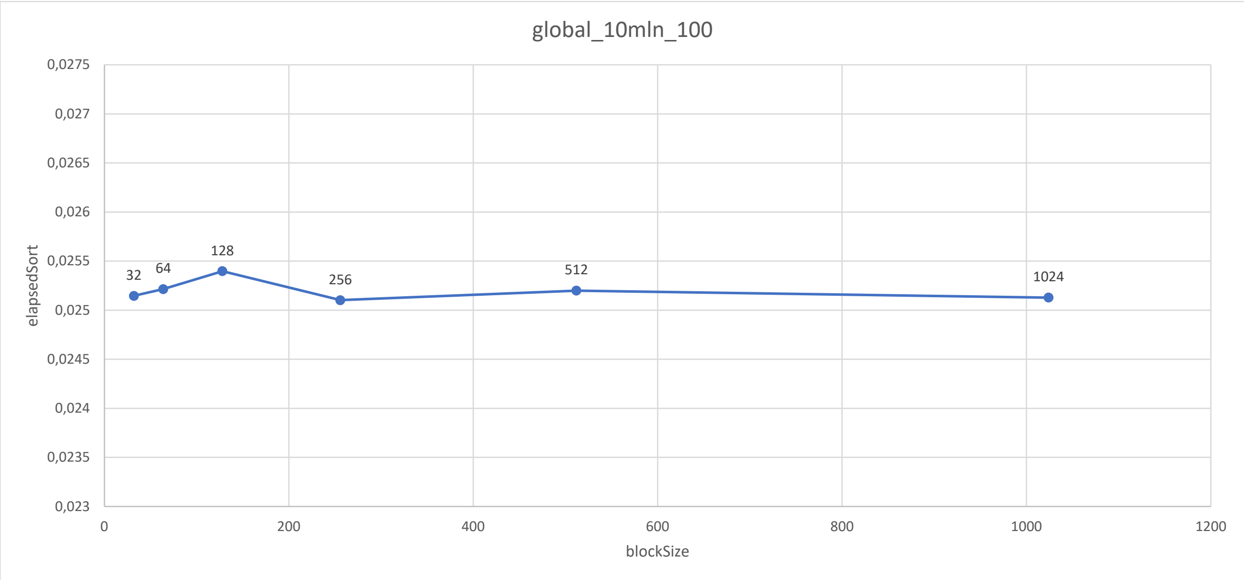
blockSize	gridSize	elapsedSort	MIPSSort
32	31250	0,00399314	3756,823
64	15625	0,0037174	4035,487
128	7813	0,00369884	4055,736
256	3907	0,003653	4106,63
512	1954	0,00369326	4061,864
1024	977	0,00370116	4053,194



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		3.814697	2.389192	[CUDA memcpy DtoH]
		3.814697	6.766403	[CUDA memcpy HtoD]
		0.003819	0.706253	[CUDA memset]
8	0			gpu_fullC(int*, int*, int)
20	0			gpu_sumC(int*, int)
9	0			
		3.814697	5.241358	[CUDA memcpy DtoH]

SIZE-10mln-RANGE-100

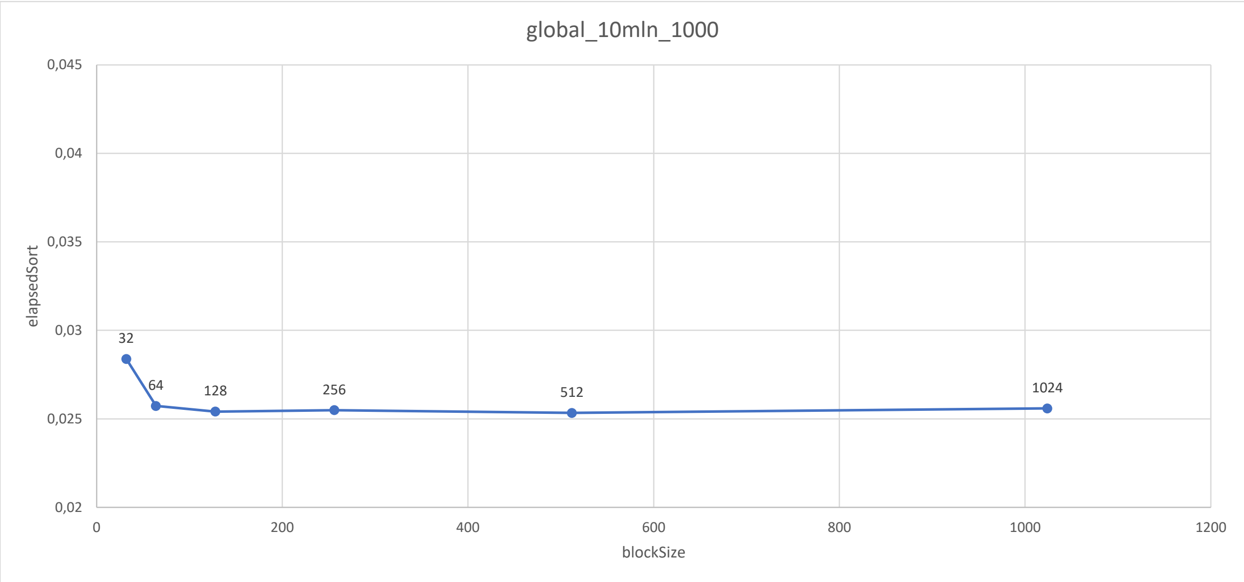
blockSize	gridSize	elapsedSort	MIPSSort
32	312500	0,02514548	5965,292
64	156250	0,02521452	5948,959
128	78125	0,02539706	5906,201
256	39063	0,02510244	5975,52
512	19532	0,02519928	5952,557
1024	9766	0,025128	5969,442



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		38.146973	1.435224	[CUDA memcpy DtoH]
		38.146973	6.402967	[CUDA memcpy HtoD]
		0.000385	0.078912	[CUDA memset]
8	0			gpu_fullC(int*, int*, int)
20	0			gpu_sumC(int*, int)
9	0			
		38.146973	6.299201	[CUDA memcpy DtoH]

SIZE-10mln-RANGE-1000

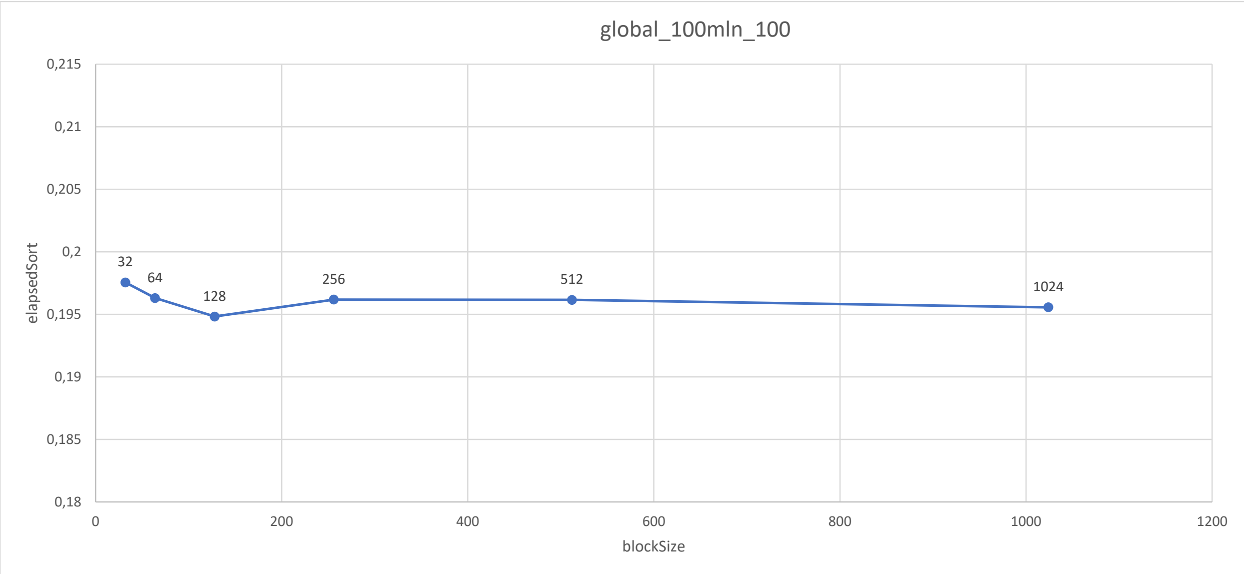
blockSize	gridSize	elapsedSort	MIPSSort
32	312500	0,02838104	5285,263
64	156250	0,0257257	5830,794
128	78125	0,02540804	5903,693
256	39063	0,02548754	5885,278
512	19532	0,02533084	5921,685
1024	9766	0,02559128	5861,421



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		38.146973	1.431185	[CUDA memcpy DtoH]
		38.146973	6.168201	[CUDA memcpy HtoD]
		0.003819	0.714919	[CUDA memset]
8	0			gpu_fullC(int*, int*, int)
20	0			gpu_sumC(int*, int)
9	0			v
		38.146973	5.468730	[CUDA memcpy DtoH]

SIZE-100mln-RANGE-100

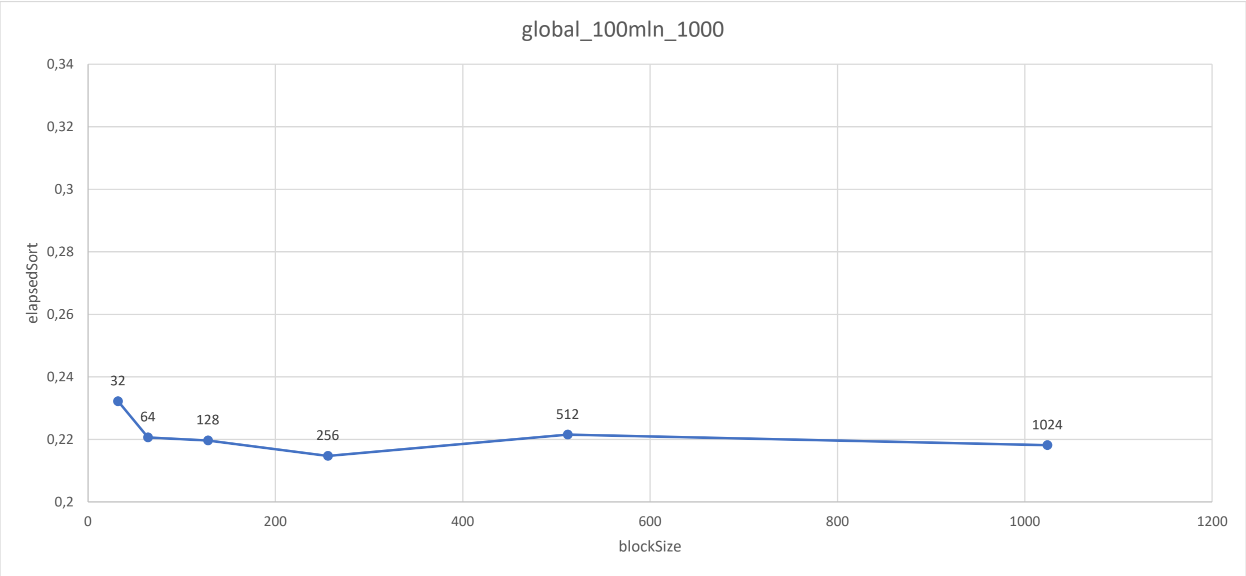
blockSize	gridSize	elapsedSort	MIPSSort
32	3125000	0,19755194	7592,941
64	1562500	0,196302	7641,288
128	781250	0,1948261	7699,174
256	390625	0,19617746	7646,139
512	195313	0,19615648	7646,957
1024	97657	0,19555886	7670,326



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		381.469727	1.424407	[CUDA memcpy DtoH]
		381.469727	6.554349	[CUDA memcpy HtoD]
		0.000385	0.081089	[CUDA memset]
8	0			gpu_fullC(int*, int*, int)
20	0			gpu_sumC(int*, int)
9	0			v
		381.469727	6.720254	[CUDA memcpy DtoH]

SIZE-100mln-RANGE-1000

blockSize	gridSize	elapsedSort	MIPSSort
32	3125000	0,23219668	6460,046
64	1562500	0,22062692	6798,813
128	781250	0,21963964	6829,374
256	390625	0,21471408	6986,041
512	195313	0,22152676	6771,197
1024	97657	0,21815504	6875,85



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		381.469727	1.438824	[CUDA memcpy DtoH]
		381.469727	6.229231	[CUDA memcpy HtoD]
		0.003819	0.742240	[CUDA memset]
8	0			gpu_fullC(int*, int*, int)
20	0			gpu_sumC(int*, int)
9	0			sv
		381.469727	6.492941	[CUDA memcpy DtoH]

Case Study n.2 – Shared Memory & Global Memory

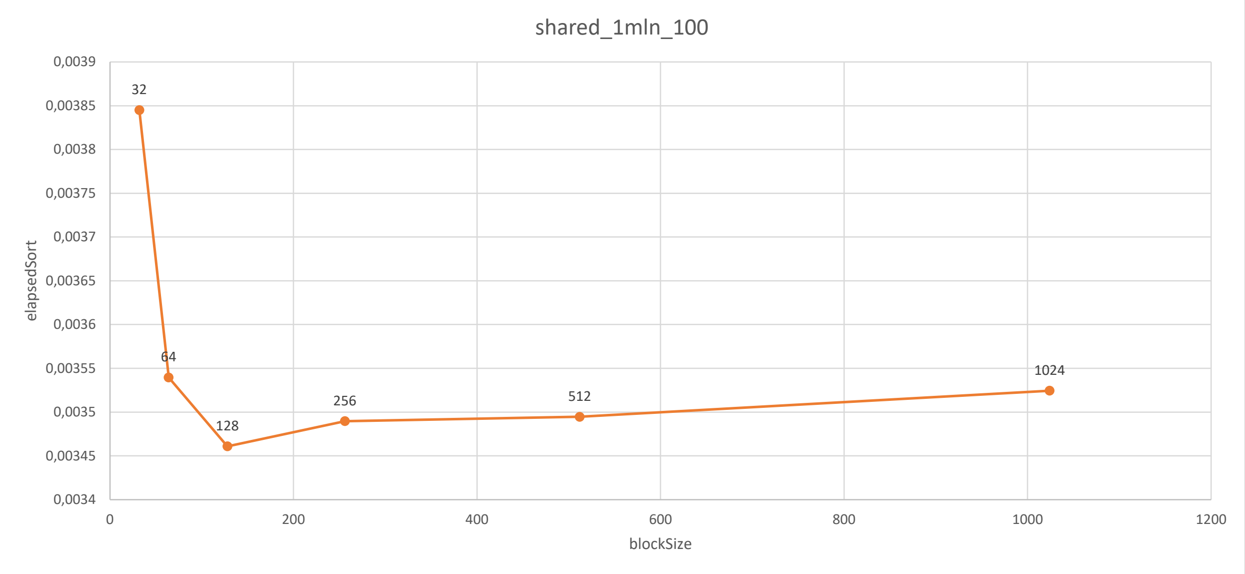
In this case study the memories exploited among global, shared and texture are global and shared.

The only kernel whose behaviour changes than Case Study n.1 is:

- **gpu_fullC**: each block has a __shared__ array, whose name is **C_shared**, dynamically allocated. Its dimension inside the shared memory of every single block is equal to **c** dimension. It is relevant to notice that for Tesla K80 the total amount of shared memory per block is 49152 bytes and an integer has a typical storage size of 4 bytes. This means that there is a limit for **c** size given by 49152 bytes / 4 bytes = 12288. This limit does not refer to the program version corresponding to Case Study n.1, but only to Case Studies n.2 and n.3, where shared memory is used. The kernel logic is characterized by a first for-loop in which, for each block, every thread initializes to 0 one or more elements inside **C_shared** (it is important to remember that every block has a different instance of **C_shared** allocated on its shared memory). Then thread index is computed thanks to the formula $blockIdx.x * blockDim.x + threadIdx.x$, so that each thread increments by 1 the right position of the instance of **C_shared** associated to the block to which the current thread belongs. In the end, there is a final for-loop that makes a sort of reduction, putting all together (in **c**) the partial results resident on the different **C_shared** instances.

SIZE-1mln-RANGE-100

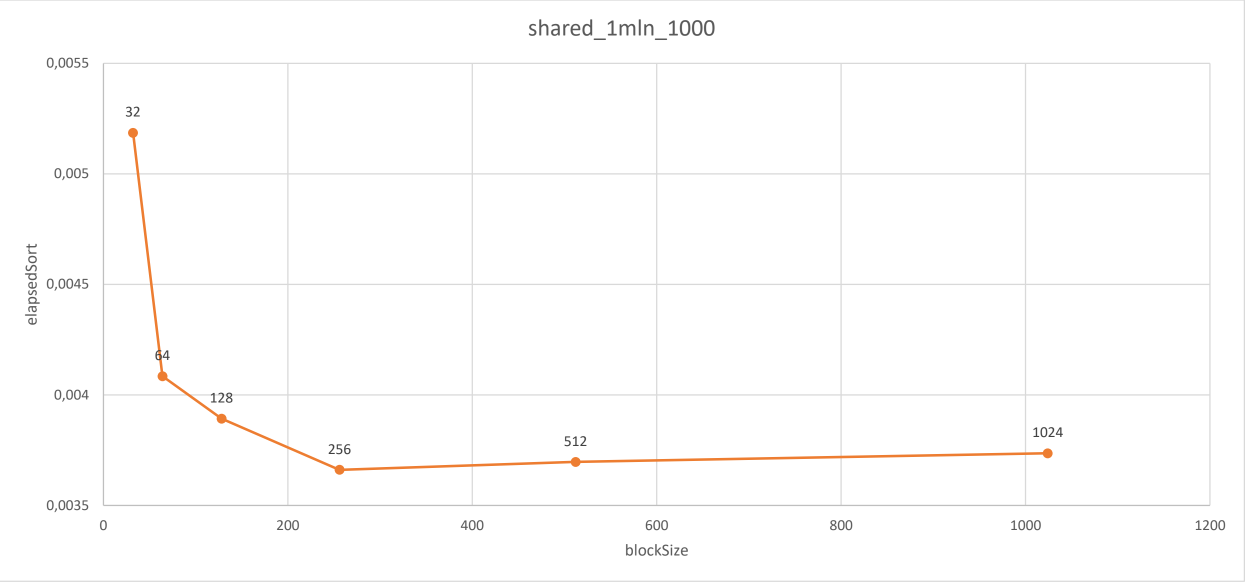
blockSize	gridSize	elapsedSort	MIPSSort
32	31250	0,00384506	6063,216
64	15625	0,00353952	6586,608
128	7813	0,00346082	6736,389
256	3907	0,00348958	6680,87
512	1954	0,00349458	6671,311
1024	977	0,00352446	6614,752



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		3.814697	2.309058	[CUDA memcpy DtoH]
		3.814697	2.018688	[CUDA memcpy HtoD]
		0.000385	0.075371	[CUDA memset]
8	404			gpu_fullC(int*, int*, int, int)
20	0			gpu_sumC(int*, int)
9	0			gpu_lastKernel(int*, int*, int*, int)
		3.814697	4.156224	[CUDA memcpy DtoH]

SIZE-1mln-range-1000

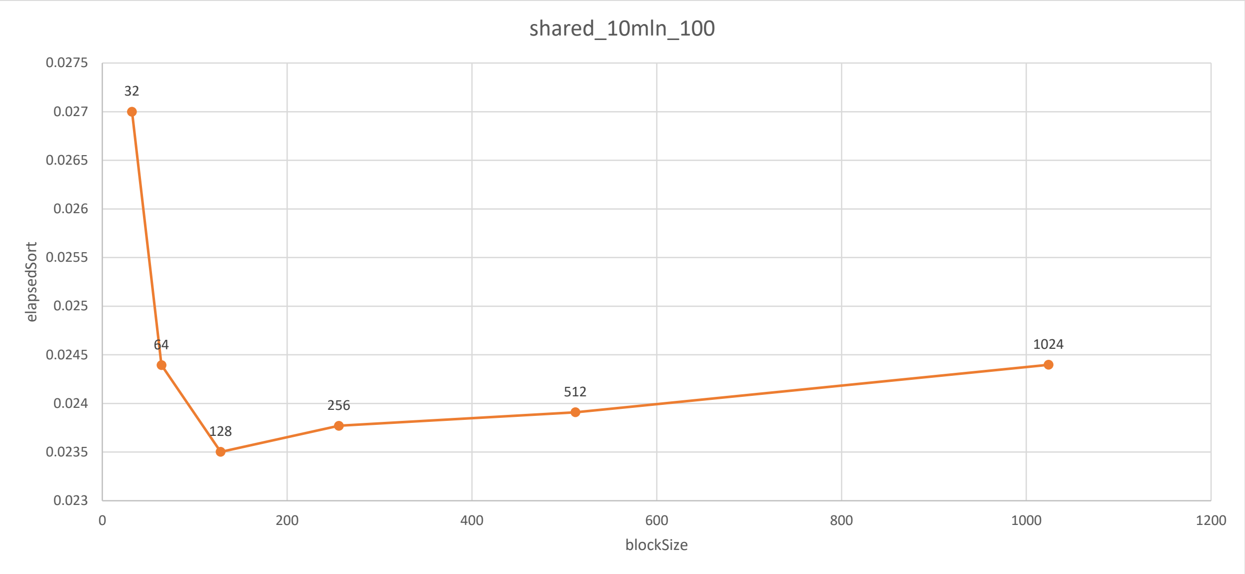
blockSize	gridSize	elapsedSort	MIPSSort
32	31250	0,0051856	15344,06
64	15625	0,0040846	19480,04
128	7813	0,0038924	20441,93
256	3907	0,00366084	21734,94
512	1954	0,00369678	21523,64
1024	977	0,0037356	21299,97



Regs	DSMEM	Size	Throughput	Name
	KB	MB	GB/s	
9	0.000000			gpu_initArray(int*, int, int, int)
		3.814697	2.072597	[CUDA memcpy DtoH]
		3.814697	6.710615	[CUDA memcpy HtoD]
		0.003819	0.665896	[CUDA memset]
8	3.910156			gpu_fullC(int*, int*, int, int)
20	0.000000			gpu_sumC(int*, int)
9	0.000000			gpu_lastKernel(int*, int*, int*, int)
		3.814697	6.253530	[CUDA memcpy DtoH]

SIZE-10mln-RANGE-100

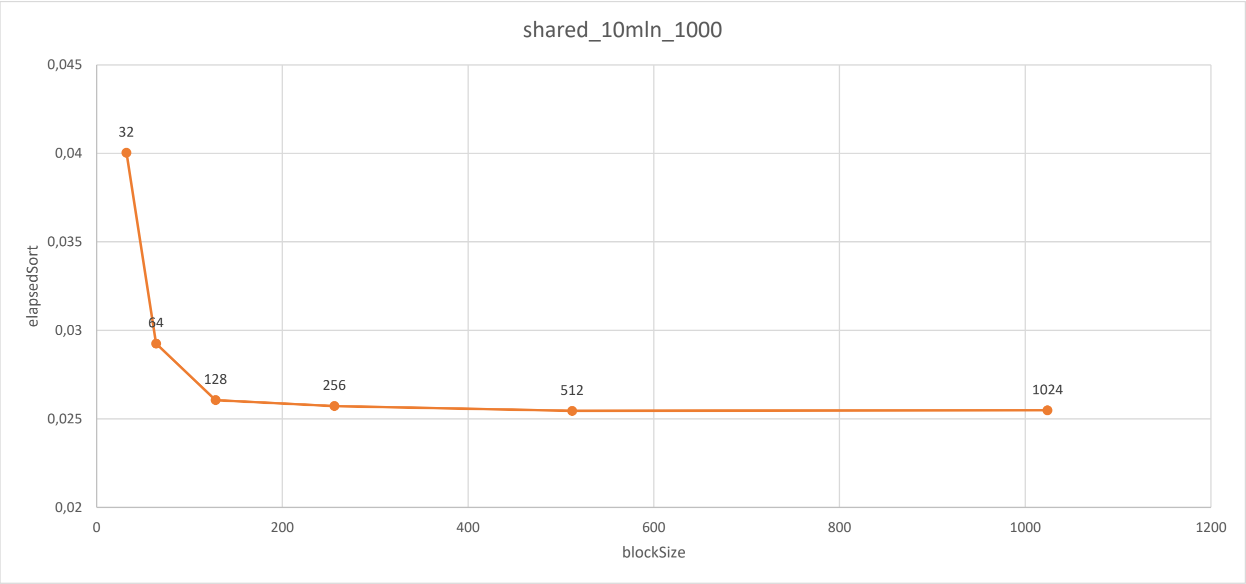
blockSize	gridSize	elapsedSort	MIPSSort
32	312500	0,0269992	8634,52
64	156250	0,02439424	9556,565
128	78125	0,02350144	9919,611
256	39063	0,02377038	9807,38
512	19532	0,02390904	9750,502
1024	9766	0,02439734	9555,351



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		38.146973	1.303461	[CUDA memcpy DtoH]
		38.146973	5.646525	[CUDA memcpy HtoD]
		0.000385	0.079446	[CUDA memset]
8	404			gpu_fullC(int*, int*, int, int)
20	0			gpu_sumC(int*, int)
9	0			gpu_lastKernel(int*, int*, int*, int)
		38.146973	5.784652	[CUDA memcpy DtoH]

SIZE-10mln-RANGE-1000

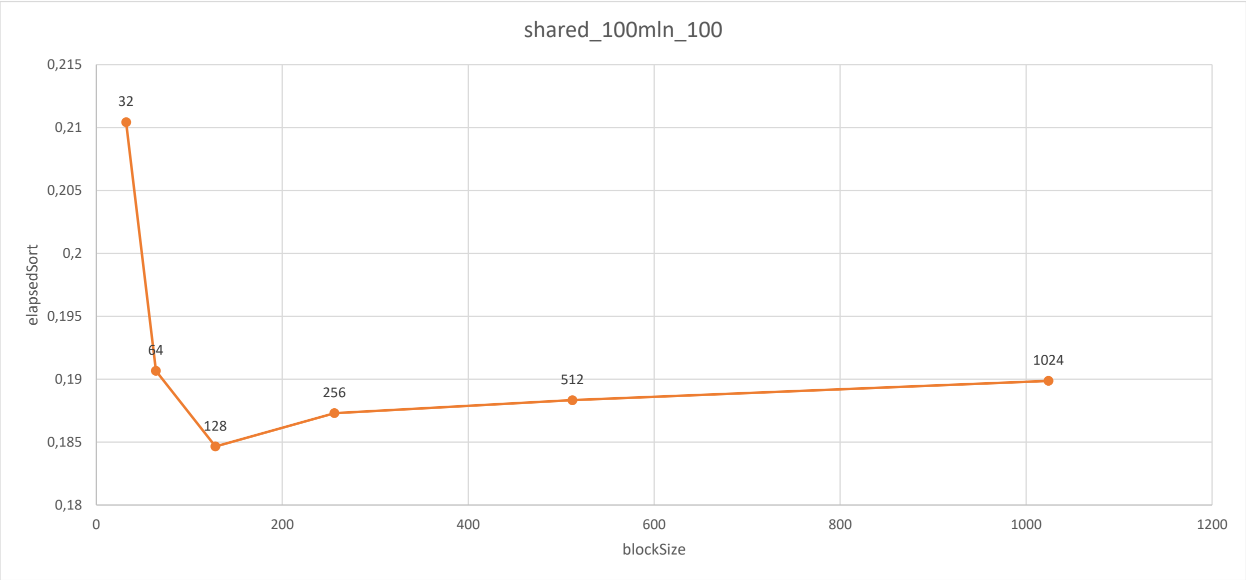
blockSize	gridSize	elapsedSort	MIPSSort
32	312500	0,04003972	19870,92
64	156250	0,02925094	27200,02
128	78125	0,02606086	30529,55
256	39063	0,02572466	30928,54
512	19532	0,02544962	31262,8
1024	9766	0,02548342	31221,33



Regs	DSMEM	Size	Throughput	Name
	KB	MB	GB/s	
9	0.000000			gpu_initArray(int*, int, int, int)
		38.146973	1.438766	[CUDA memcpy DtoH]
		38.146973	6.345622	[CUDA memcpy HtoD]
		0.003819	0.714919	[CUDA memset]
8	3.910156			gpu_fullC(int*, int*, int, int)
20	0.000000			gpu_sumC(int*, int)
9	0.000000			gpu_lastKernel(int*, int*, int*, int)
		38.146973	5.948400	[CUDA memcpy DtoH]

SIZE-100mln-RANGE-100

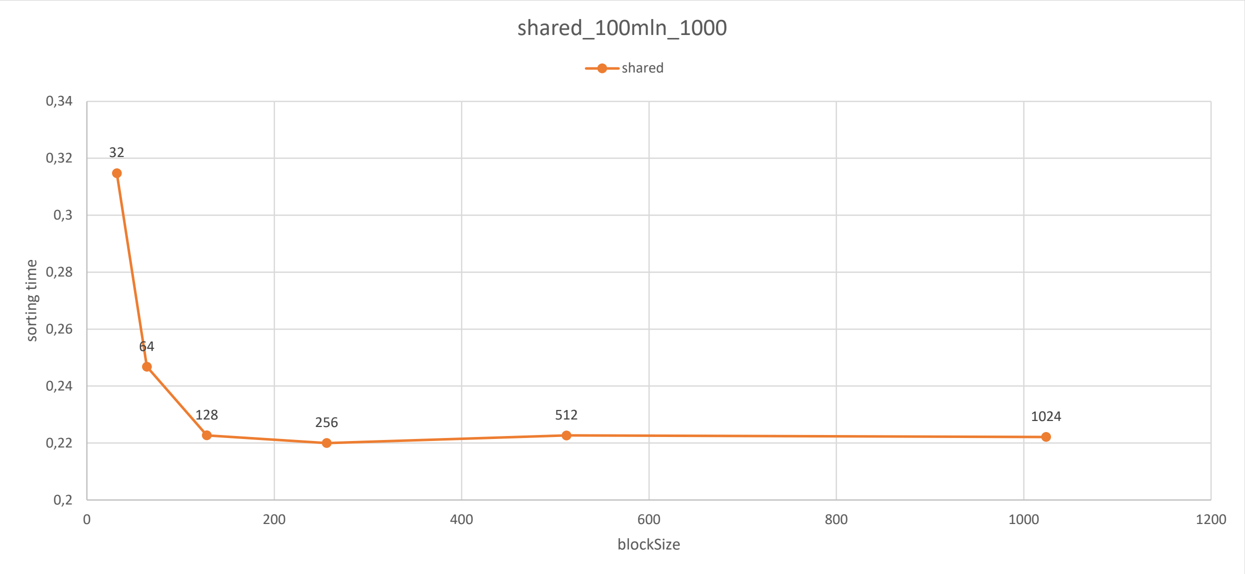
blockSize	gridSize	elapsedSort	MIPSSort
32	3125000	0,21042076	11078,99
64	1562500	0,19066098	12227,2
128	781250	0,1846503	12625,22
256	390625	0,1872984	12446,72
512	195313	0,18832868	12378,63
1024	97657	0,1898623	12278,64



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		381.469727	1.439124	[CUDA memcpy DtoH]
		381.469727	6.567264	[CUDA memcpy HtoD]
		0.000385	0.081089	[CUDA memset]
8	404			gpu_fullC(int*, int*, int, int)
20	0			gpu_sumC(int*, int)
9	0			gpu_lastKernel(int*, int*, int*, int)
		381.469727	6.797036	[CUDA memcpy DtoH]

SIZE-100mln-RANGE-1000

blockSize	gridSize	elapsedSort	MIPSSort
32	3125000	0,31476328	25276,94
64	1562500	0,24678498	32239,61
128	781250	0,2226731	35730,64
256	390625	0,2199694	36169,81
512	195313	0,22264082	35735,82
1024	97657	0,22211214	35820,88



Regs	DSMEM	Size	Throughput	Name
	KB	MB	GB/s	
9	0.000000			gpu_initArray(int*, int, int, int)
		381.469727	1.408998	[CUDA memcpy DtoH]
		381.469727	6.466210	[CUDA memcpy HtoD]
		0.003819	0.742240	[CUDA memset]
8	3.910156			gpu_fullC(int*, int*, int, int)
20	0.000000			gpu_sumC(int*, int)
9	0.000000			gpu_lastKernel(int*, int*, int*, int)
		381.469727	6.513721	[CUDA memcpy DtoH]

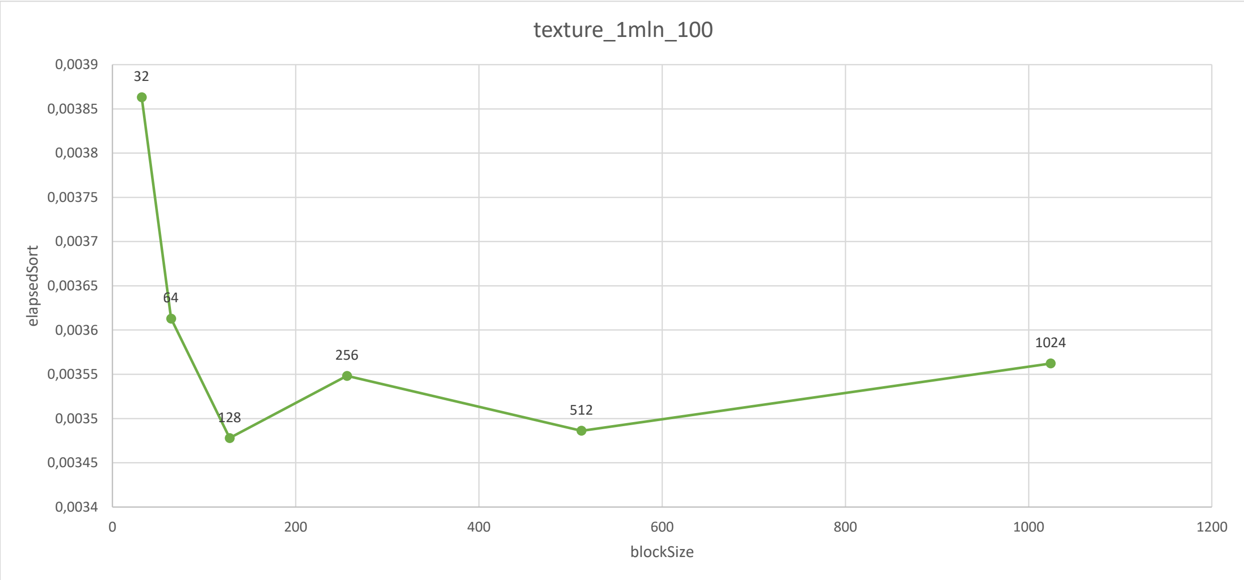
Case Study n.3 – Texture Memory & Shared Memory & Global Memory

In this case study all the three different types of memories are exploited: global, shared and texture.

To understand the few differences in source code between the program versions corresponding to Case Studies n.2 and n.3, it is fundamental to know that texture is a read only memory. Texture memory, for this reason, is used to manage accesses to the elements of array **a** after its creation.

SIZE-1mln-RANGE-100

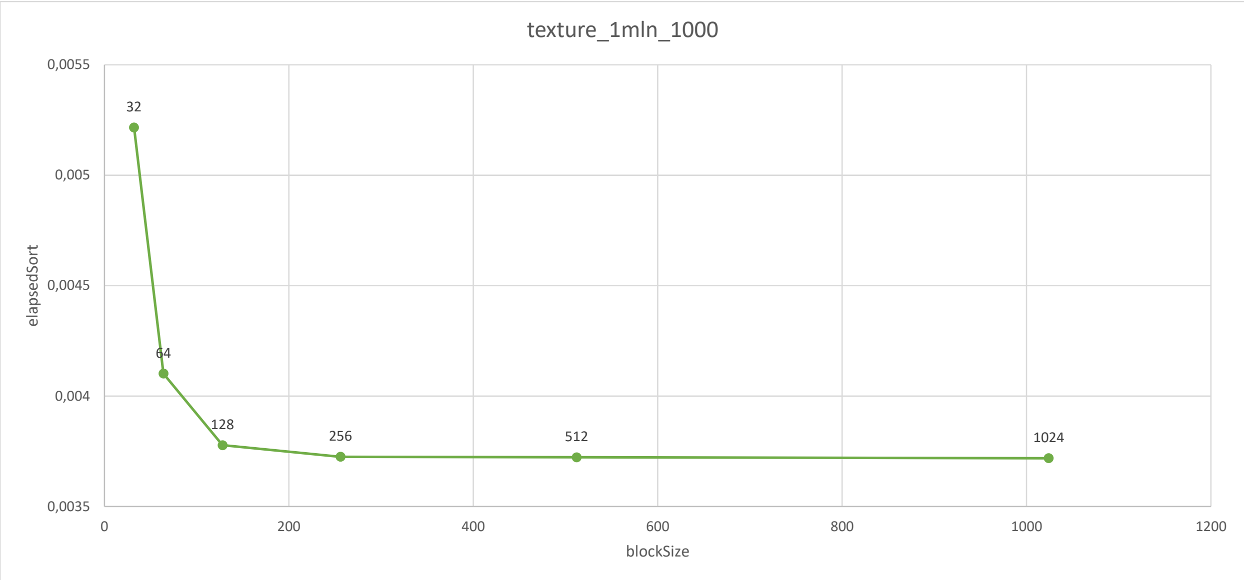
blockSize	gridSize	elapsedSort	MIPSSort
32	31250	0,00386312	4999,438
64	15625	0,00361294	5345,627
128	7813	0,00347788	5553,219
256	3907	0,0035482	5443,162
512	1954	0,0034862	5539,966
1024	977	0,00356218	5421,8



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		3.814697	2.027831	[CUDA memcpy DtoH]
		3.814697	7.025695	[CUDA memcpy HtoD]
		0.000385	0.079446	[CUDA memset]
8	404			gpu_fullC(int*, int*, int, int)
20	0			gpu_sumC(int*, int)
8	0			gpu_lastKernel(int*, int*, int*, int)
		3.814697	5.536493	[CUDA memcpy DtoH]

SIZE-1mln-range-1000

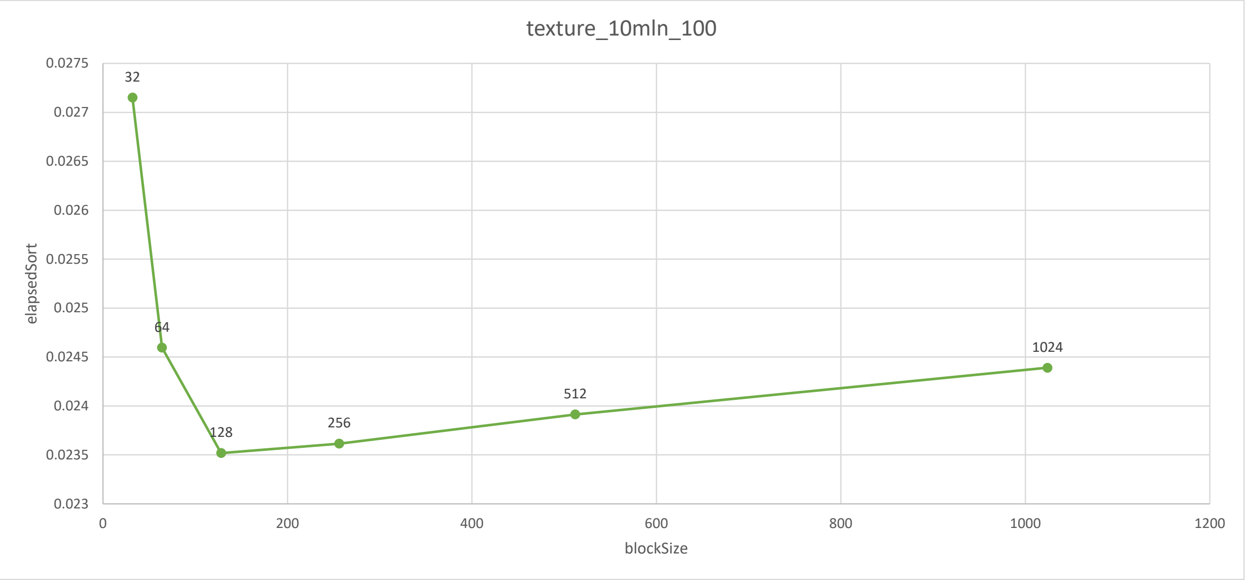
blockSize	gridSize	elapsedSort	MIPSSort
32	31250	0,00521582	14488,26
64	15625	0,00410168	18423,71
128	7813	0,00377716	20006,61
256	3907	0,00372466	20288,6
512	1954	0,00372248	20300,49
1024	977	0,0037183	20323,31



Regs	DSMEM	Size	Throughput	Name
	KB	MB	GB/s	
9	0.000000			gpu_initArray(int*, int, int, int)
		3.814697	1.955187	[CUDA memcpy DtoH]
		3.814697	6.992357	[CUDA memcpy HtoD]
		0.003819	0.701998	[CUDA memset]
8	3.910156			gpu_fullC(int*, int*, int, int)
20	0.000000			gpu_sumC(int*, int)
8	0.000000			gpu_lastKernel(int*, int*, int*, int)
		3.814697	6.531037	[CUDA memcpy DtoH]

SIZE-10mln-RANGE-100

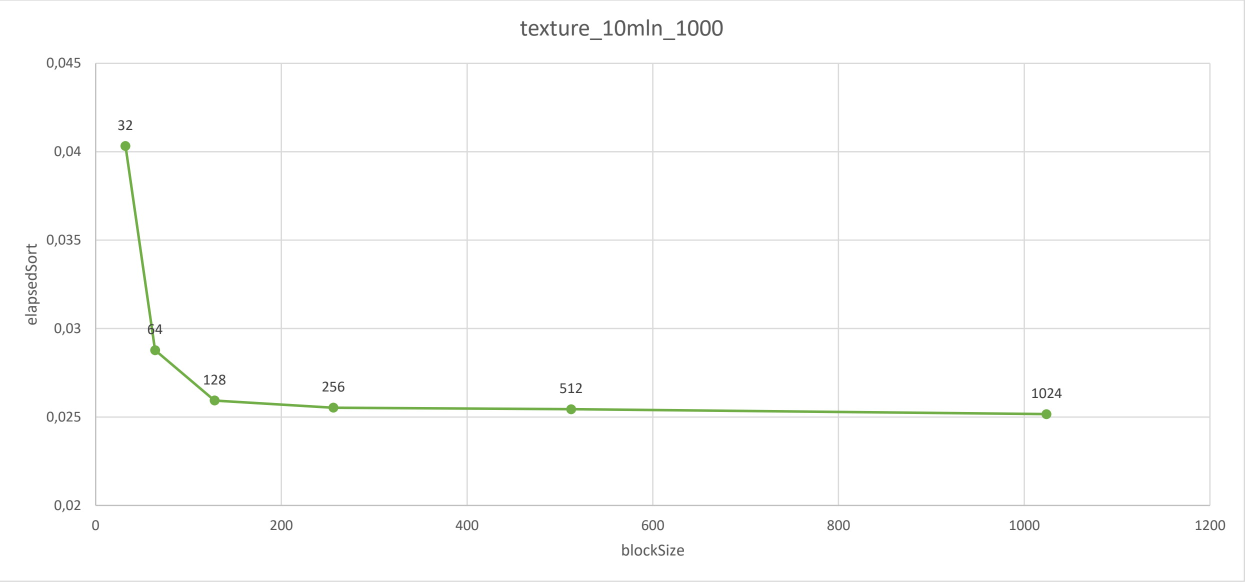
blockSize	gridSize	elapsedSort	MIPSSort
32	312500	0,02715078	7113,061
64	156250	0,02459618	7851,835
128	78125	0,02351836	8211,676
256	39063	0,02361468	8178,182
512	19532	0,02391342	8076,015
1024	9766	0,02438974	7918,294



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		38.146973	1.421172	[CUDA memcpy DtoH]
		38.146973	6.606220	[CUDA memcpy HtoD]
		0.000385	0.077355	[CUDA memset]
8	404			gpu_fullC(int*, int*, int, int)
20	0			gpu_sumC(int*, int)
8	0			gpu_lastKernel(int*, int*, int*, int)
		38.146973	5.992986	[CUDA memcpy DtoH]

SIZE-10mln-RANGE-1000

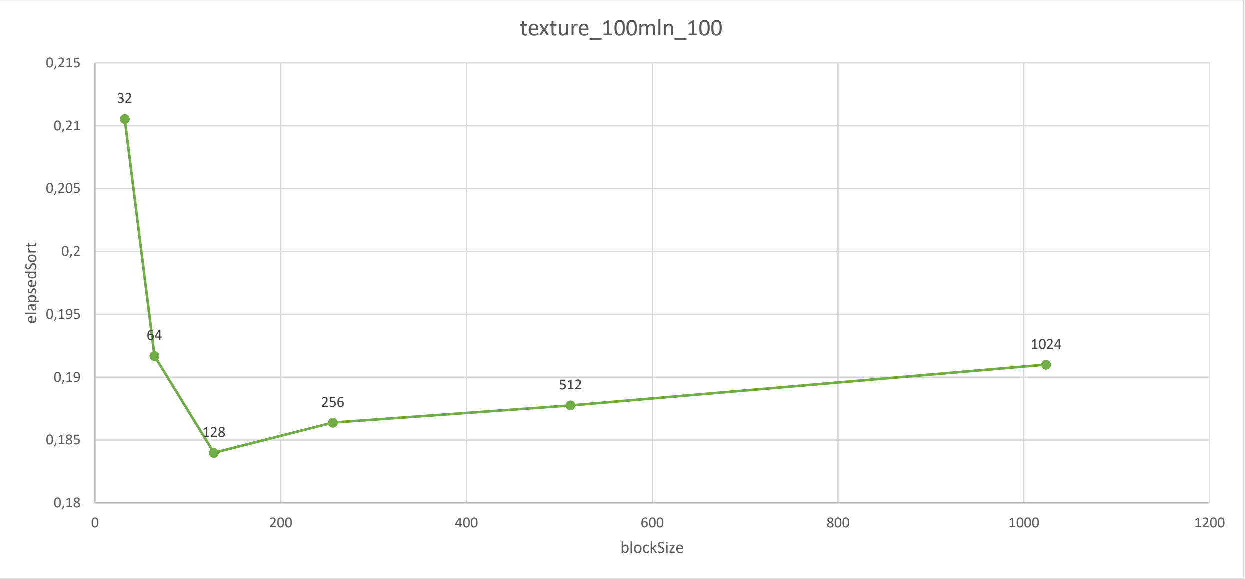
blockSize	gridSize	elapsedSort	MIPSSort
32	312500	0,04032768	18737,16
64	156250	0,02877662	26258,34
128	78125	0,02593208	29138,67
256	39063	0,02552472	29603,7
512	19532	0,02543894	29703,53
1024	9766	0,02516184	30030,64



Regs	DSMEM	Size	Throughput	Name
	KB	MB	GB/s	
9	0.000000			gpu_initArray(int*, int, int, int)
		38.146973	1.319811	[CUDA memcpy DtoH]
		38.146973	6.196701	[CUDA memcpy HtoD]
		0.003819	0.706253	[CUDA memset]
8	3.910156			gpu_fullC(int*, int*, int, int)
20	0.000000			gpu_sumC(int*, int)
8	0.000000			gpu_lastKernel(int*, int*, int*, int)
		38.146973	5.763433	[CUDA memcpy DtoH]

SIZE-100mln-RANGE-100

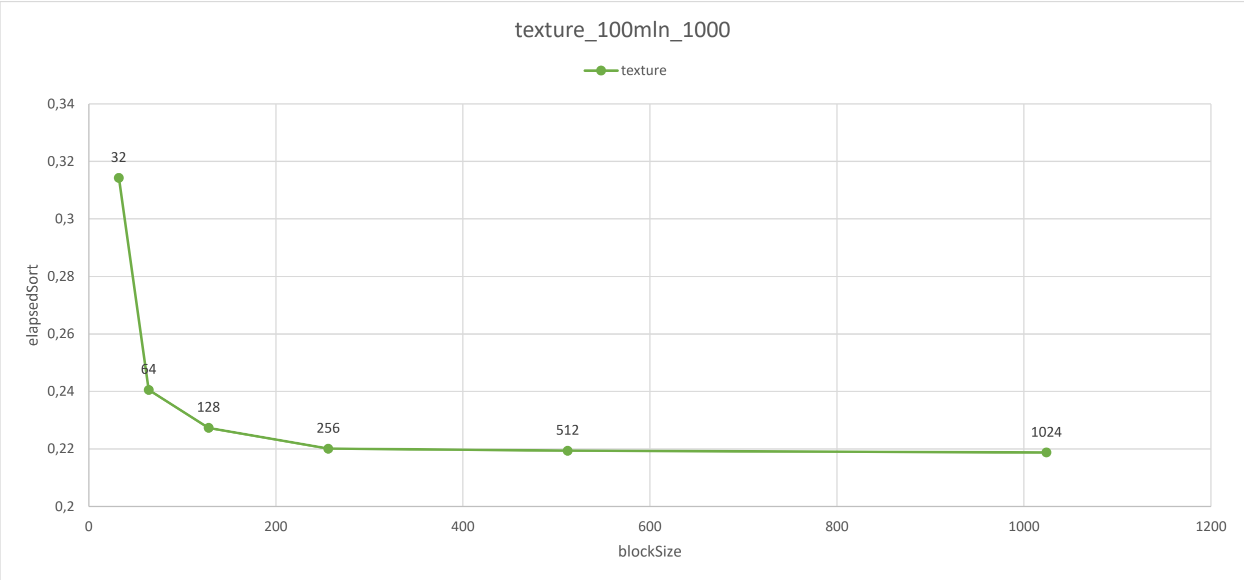
blockSize	gridSize	elapsedSort	MIPSSort
32	3125000	0,21054818	9172,486
64	1562500	0,1916854	10075,1
128	781250	0,1839804	10497,04
256	390625	0,18638236	10361,76
512	195313	0,18776036	10285,72
1024	97657	0,19099594	10111,47



Regs	DSMEM	Size	Throughput	Name
	B	MB	GB/s	
9	0			gpu_initArray(int*, int, int, int)
		381.469727	1.494610	[CUDA memcpy DtoH]
		381.469727	6.378059	[CUDA memcpy HtoD]
		0.000385	0.082802	[CUDA memset]
8	404			gpu_fullC(int*, int*, int, int)
20	0			gpu_sumC(int*, int)
8	0			gpu_lastKernel(int*, int*, int*, int)
		381.469727	6.376637	[CUDA memcpy DtoH]

SIZE-100mln-RANGE-1000

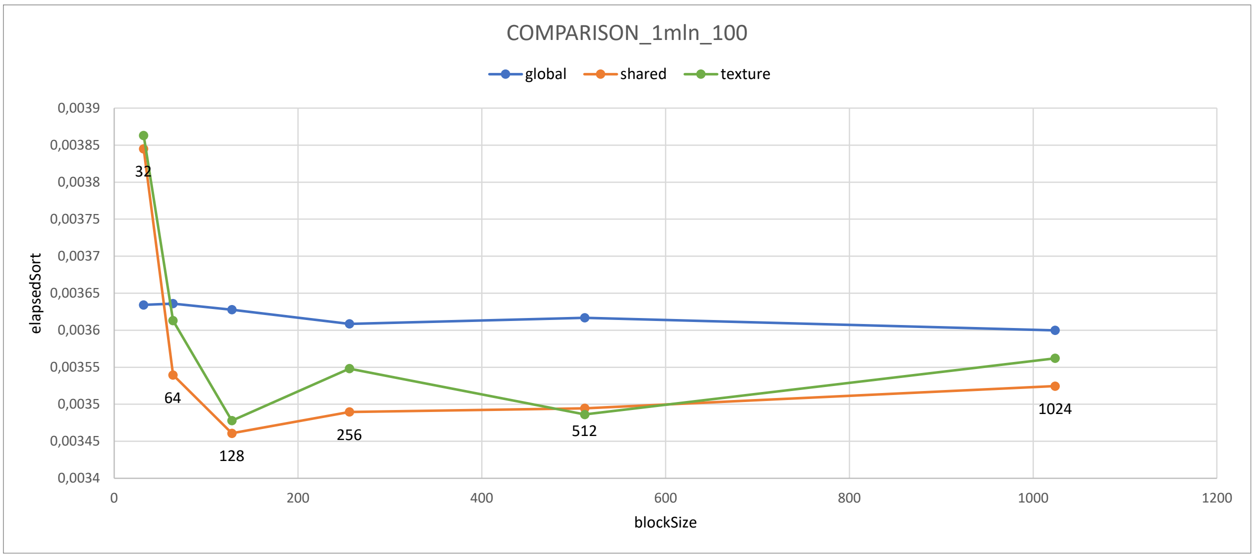
blockSize	gridSize	elapsedSort	MIPSSort
32	3125000	0,31428054	24043,01
64	1562500	0,24053102	31414,87
128	781250	0,22732904	33239,27
256	390625	0,22008558	34333,24
512	195313	0,21935964	34446,86
1024	97657	0,21877992	34538,14



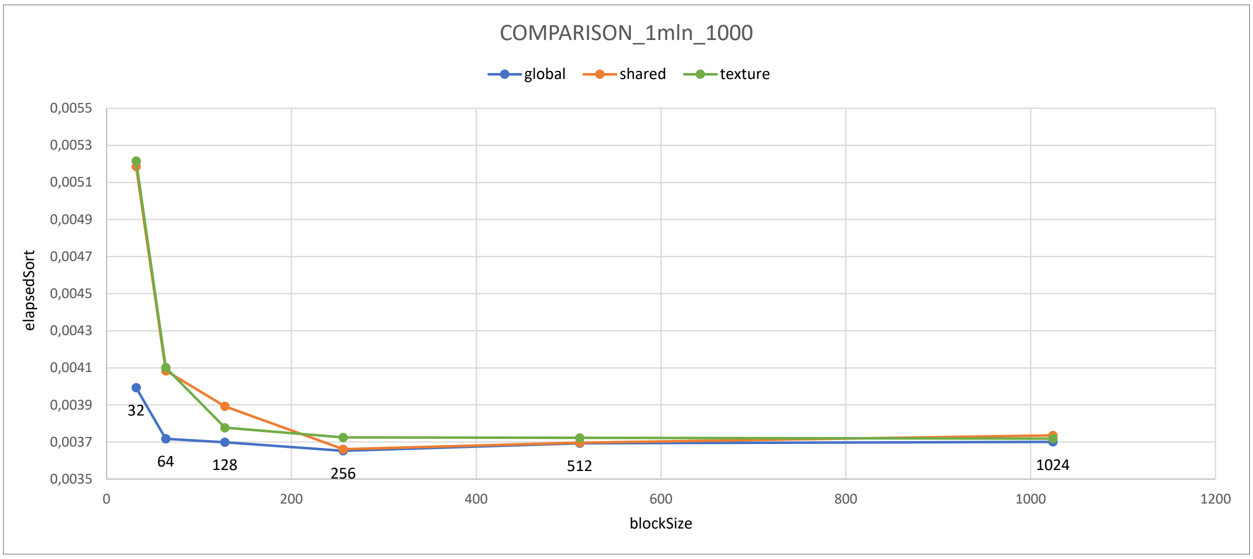
Regs	DSMEM	Size	Throughput	Name
	KB	MB	GB/s	
9	0.000000			gpu_initArray(int*, int, int, int)
		381.469727	1.425454	[CUDA memcpy DtoH]
		381.469727	6.347664	[CUDA memcpy HtoD]
		0.003819	0.706253	[CUDA memset]
8	3.910156			gpu_fullC(int*, int*, int, int)
20	0.000000			gpu_sumC(int*, int)
8	0.000000			gpu_lastKernel(int*, int*, int*, int)
		381.469727	6.492567	[CUDA memcpy DtoH]

Case Studies COMPARISONS

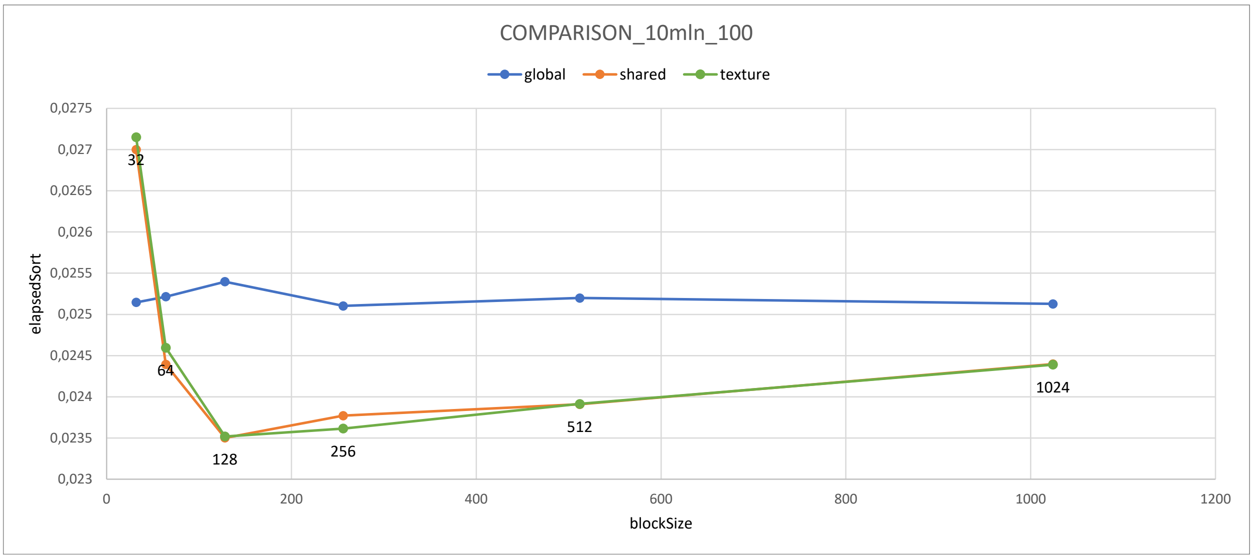
SIZE-1mln-RANGE-100



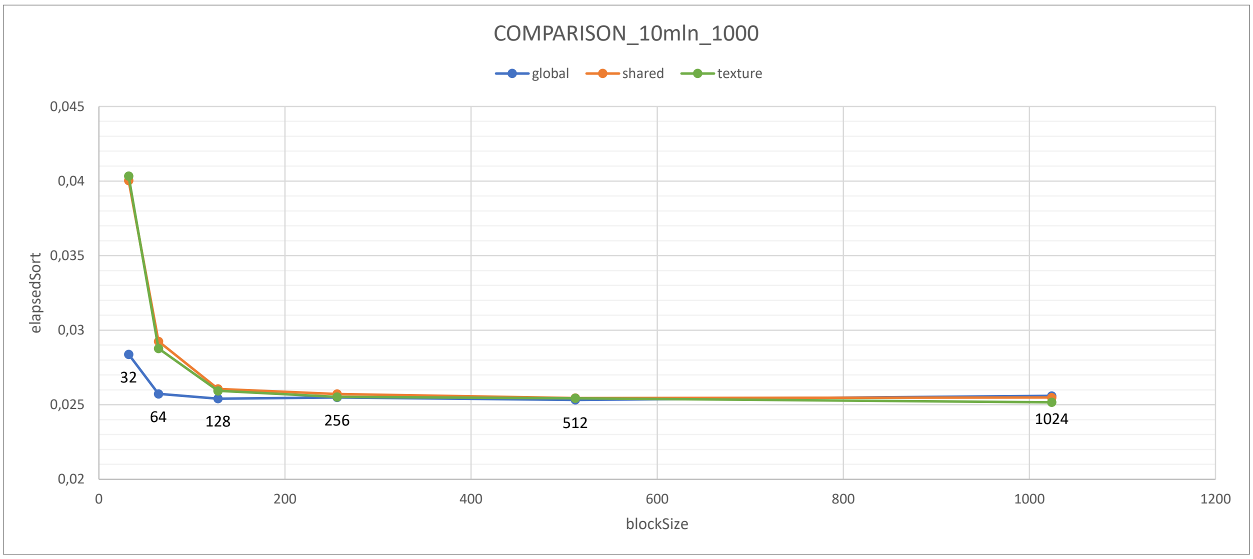
SIZE-1mln-RANGE-1000



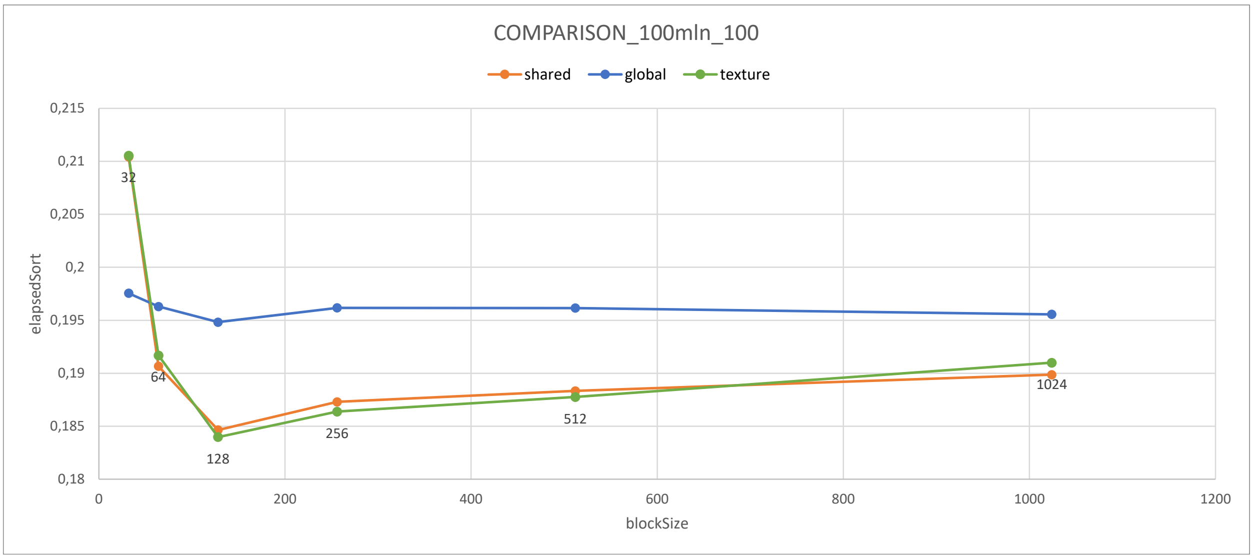
SIZE-10mln-RANGE-100



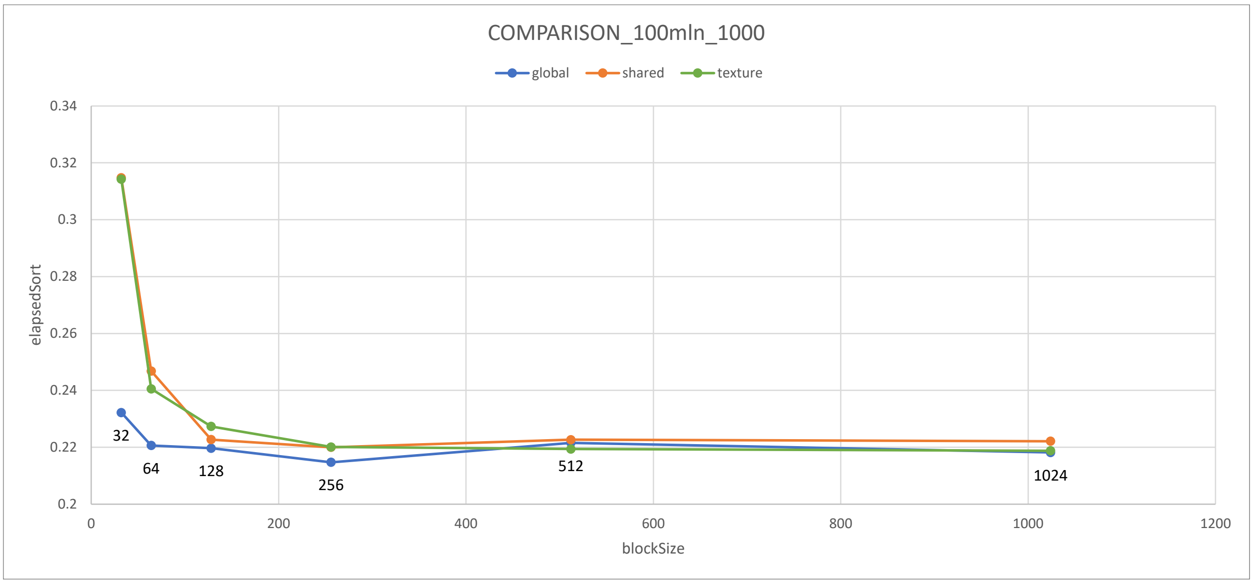
SIZE-10mln-RANGE-1000



SIZE-100mln-RANGE-100



SIZE-100mln-RANGE-1000



Considerations

1. In the most of cases, regardless of the particular Case Study, the worst performances are obtained when block size is equal to 32 and 64. This happens because these block sizes, unlike the others, do not guarantee full SMs occupancy in terms of exploited threads.
2. Considering each Case Study, it is possible to notice that both increasing the size and increasing the range determine a greater elapsed sort, as expected.
3. Considering Case Studies n.2 and n.3, where Shared Memory is exploited, fixing the size, a range increasing determines a relevant MIPS increasing. This happens because each block has an independent C_shared and the number of integer operations performed on

it depends on its length, given by range. To better understand it, it is possible to look at the first and the last for-loop in kernel gpu_fullC.

4. It is interesting to observe that the kernel that requires more registers (20) is gpu_sumC. Though this kernel is launched on one single thread, so it does not create any problem in terms of available registers per thread.
5. In the most of cases both gpu_initArray and gpu_lastKernel use 9 registers. This data is precious to determine the maximum number of allocable blocks on a SM. The logic is simple: the maximum number of registers per SM (128K for Tesla K80) must be divided by the total number of registers you need (in this case it is equal to $9 * blockSize$). For every blockSize considered, the number of allocable blocks per SM is never less than the one obtained with the analysis conducted in Preliminary Considerations. This means that, in the Case Studies considered, the number of registers per SM does not represent an obstacle.
6. Observing the trend of the graphs, it is possible to observe that Case Studies n.2 and n.3, where shared memory is exploited, always present a pick for a number of threads per block equal to 32. This can be explained by considering the two for-loops in these versions. Thanks to these, if the number of threads per block is less than the number of elements in array C_shared, each thread executes the particular operation on more than one element of the array and receives the array position on which operates by the sum $i += blockDim.x$. This means that the less the number of threads per block is, the greater the number of sums produced by for-loops is.
7. The presence of the just analysed for-loops can also explain another observation. In fact, it has been noted a performance improvement of Case Studies n.2 and n.3, where shared memory is exploited, compared to Case Study n.1, where only global memory is exploited, when range is equal to 100 but not when range is equal to 1000. This happens because a range increase obviously determines a C_shared length increase and for this, the number of operations contained in for-loops increases, making them slower.
8. It is clear that global memory, especially in the case in which range is 100, has a very high latency compared to the shared memory, whose accesses are surely faster, due to architectural characteristics. Shared memory latency, in fact, is less than L1 cache, L2 cache and Global Memory latencies. Moreover, all accesses to Global Memory pass through L2 cache, which is in common to all the SMs. L1 cache, instead, is private for each SM (so there are multiple L1 caches).
9. The use of Texture Memory, in all Case Studies, does not seem to improve the performances. This is probably because there are not many consecutive accesses to the unsorted array elements.

Further Analysis

Other interesting analysis can be done to study in deep the behaviour of a GPU when executing an algorithm in parallel.

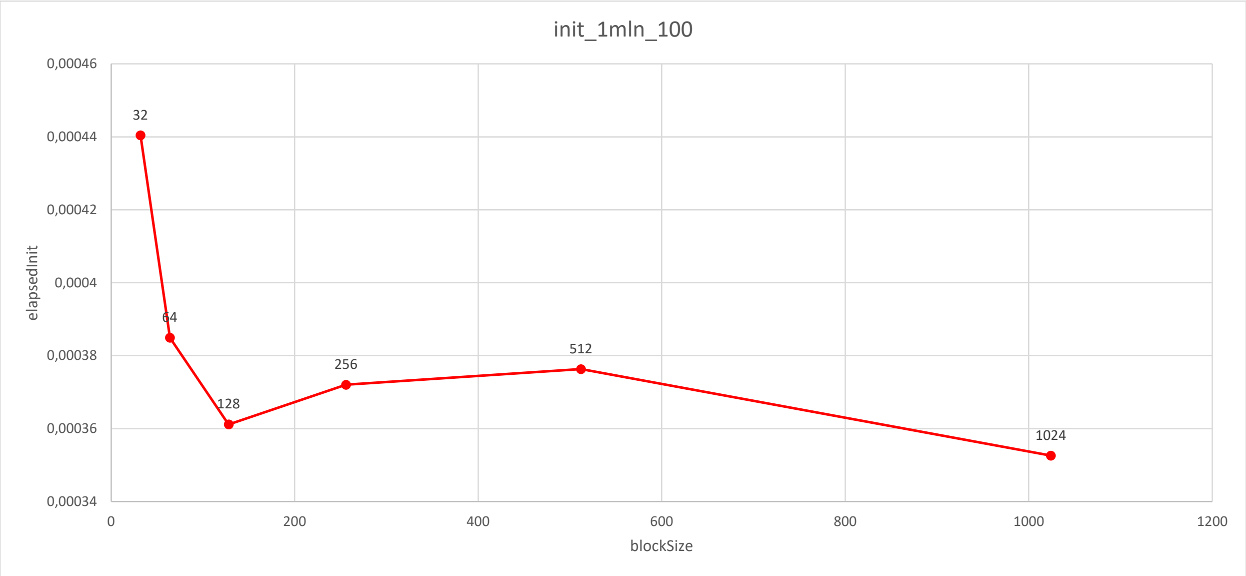
Initialization

The very first section of all the different program versions is always the same and consists of the unsorted array random initialization, performed by the function **initArray()**, which recalls the kernel **gpu_initArray**.

Analysing the differences among the execution times of this kernel when the input change is a good way to better understand the benefits of using a GPU to perform easy operations on a large amount of data and to clarify other technical aspects.

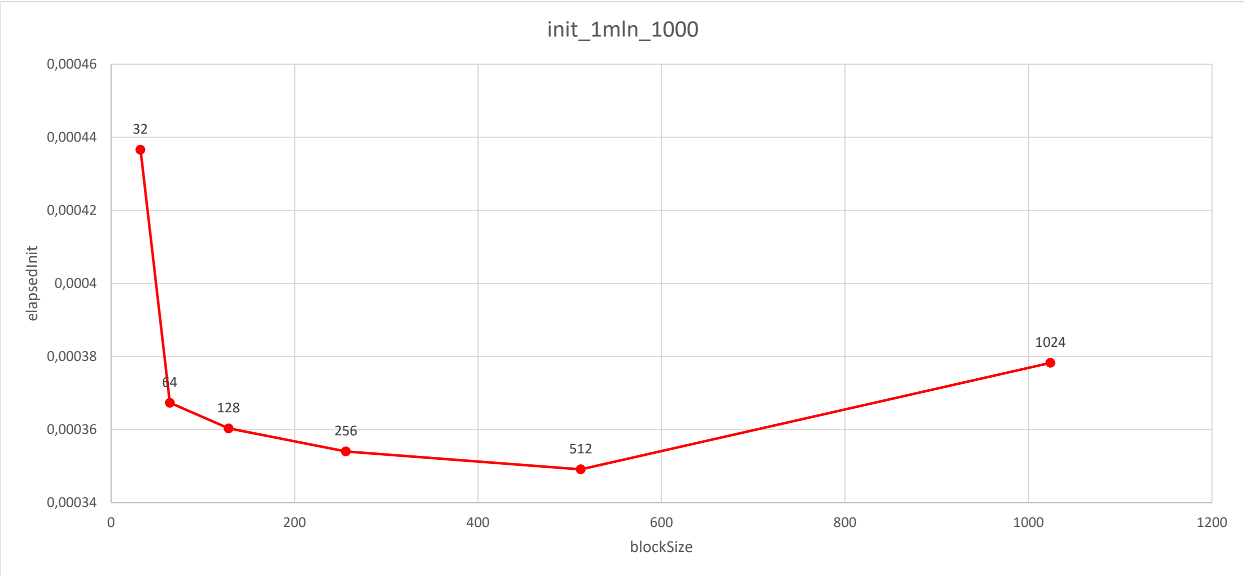
SIZE-1mln-RANGE-100

blockSize	gridSize	elapsedInit
32	31250	0,0004404
64	15625	0,0003849
128	7813	0,00036114
256	3907	0,00037202
512	1954	0,0003763
1024	977	0,0003526



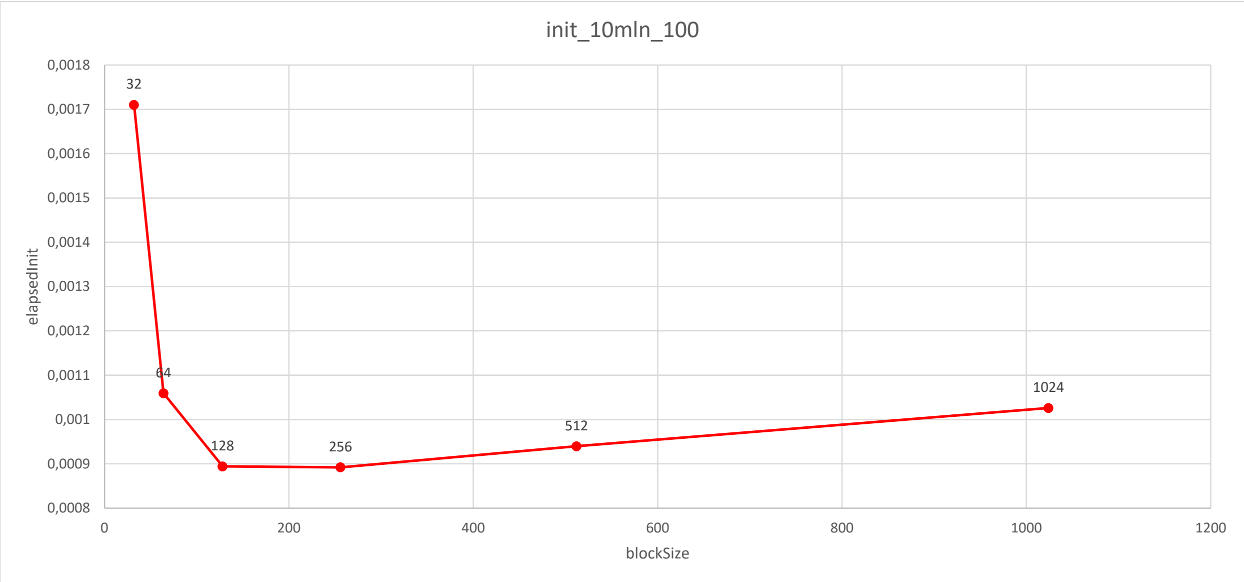
SIZE-1mln-range-1000

blockSize	gridSize	elapsedInit
32	31250	0,0004366
64	15625	0,00036732
128	7813	0,0003603
256	3907	0,000354
512	1954	0,0003491
1024	977	0,00037828



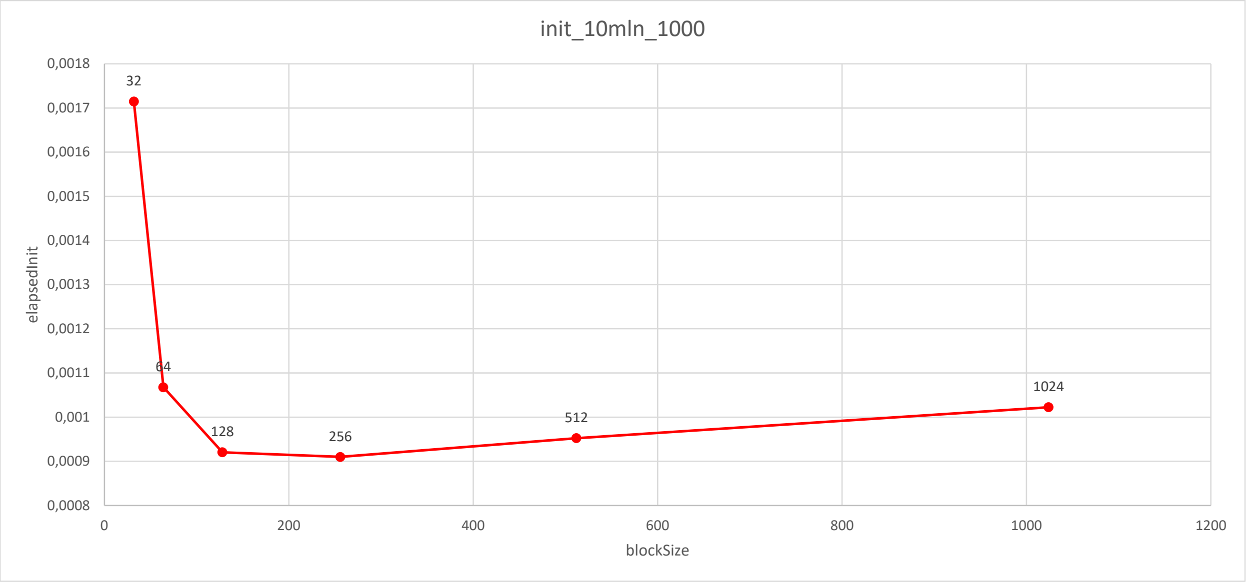
SIZE-10mln-RANGE-100

blockSize	gridSize	elapsedInit
32	312500	0,00171026
64	156250	0,00105918
128	78125	0,00089422
256	39063	0,00089198
512	19532	0,00093948
1024	9766	0,0010258



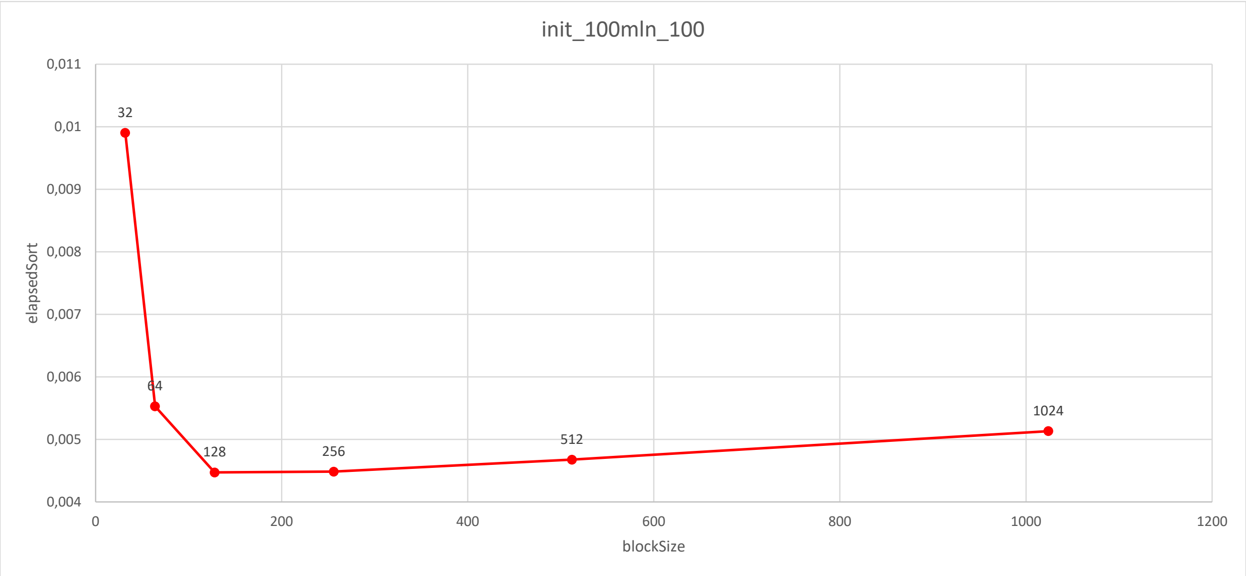
SIZE-10mln-RANGE-1000

blockSize	gridSize	elapsedInit
32	312500	0,00171452
64	156250	0,00106764
128	78125	0,00092024
256	39063	0,00090984
512	19532	0,00095242
1024	9766	0,00102236



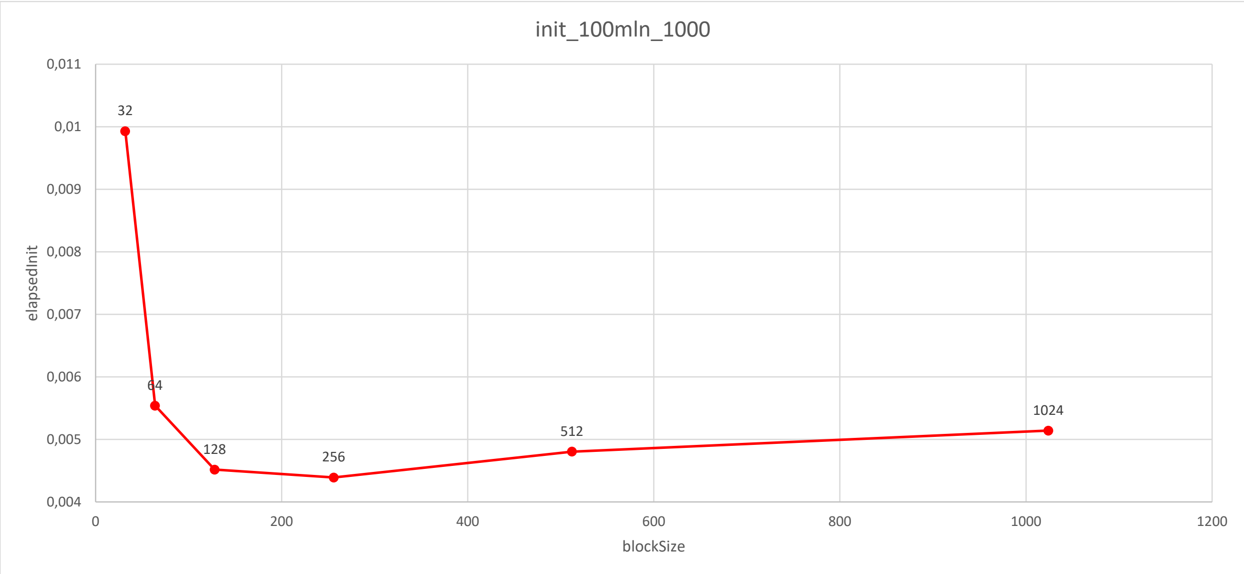
SIZE-100mln-RANGE-100

blockSize	gridSize	elapsedInit
32	3125000	0,00990122
64	1562500	0,00552686
128	781250	0,00447122
256	390625	0,00448402
512	195313	0,00467506
1024	97657	0,00513142

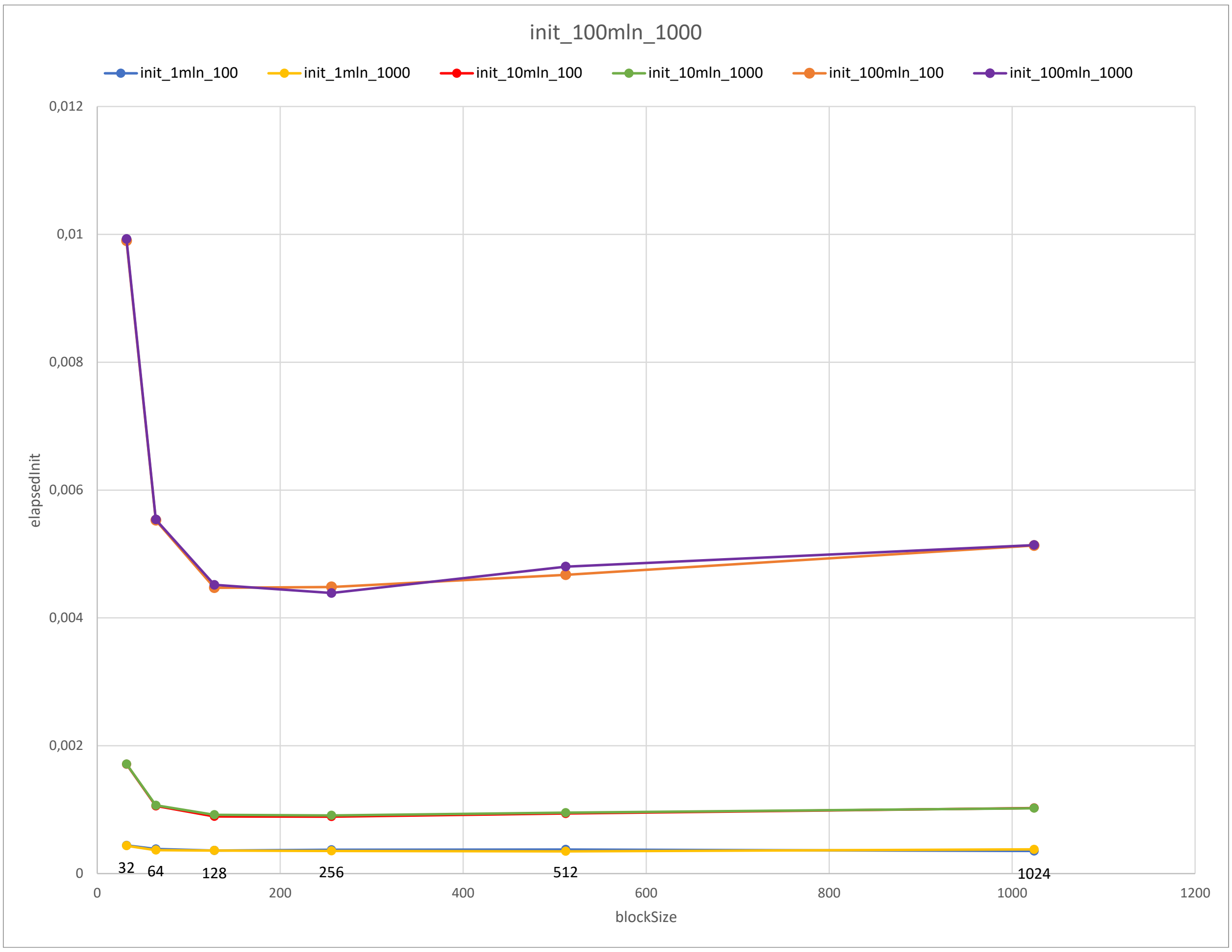


SIZE-100mln-RANGE-1000

blockSize	gridSize	elapsedInit
32	3125000	0,00992896
64	1562500	0,00553914
128	781250	0,00451638
256	390625	0,00438896
512	195313	0,00480276
1024	97657	0,00513934



Initialization COMPARISON



Considerations

The worst performances, regardless of size and range values, are always got when the block size is less than 128, which is very easy to understand knowing that block sizes of 32 and 64 do not guarantee full SMs occupancy in term of exploited threads. For greater block sizes the behaviour is better and almost stable.

Another thing to notice is that, fixing array size, changing the range does not influence, as expected, the elapsed init time, which actually depends only on array size.

Obviously the elapsed init time grows when the array size becomes greater. In particular, fixing the range to 100 (for range equal to 1000 the results are practically the same),

the average init times are as follows:

- SIZE-1mln-avgInit: 0,000381227s
- SIZE-10mln-avgInit: 0,00108682s
- SIZE-100mln-avgInit: 0,0056983s

If the execution was sequentially, SIZE-100mln-avgInit would have been approximately equal to 10 times SIZE-10mln-avgInit and 100 times SIZE-1mln-avgInit. Thanks to parallel execution provided by GPU the situation changes drastically.

In fact:

SIZE-10mln-avgInit = 2,85 * SIZE-1mln-avgInit (much less than 10 times)

SIZE-100mln-avgInit = 14.95 * SIZE-1mln-avgInit (much less than 100 times)

SIZE-100mln-avgInit = 5,24 * SIZE-10mln-avgInit (half of 10 times)

This is a great result that attests the great advantages of parallel computing on GPUs.

API

Public Docs also available here

Kernels

Type	Name
__global__ void	**gpu_initArray**(int *arrayA, int n, int range, int seed) *This is the kernel that creates random numbers and insert them in 'arrayA'.*
__global__ void	**gpu_fullC**(int *arrayA, int *arrayC, int n) *This is the kernel that fulls 'arrayC' adding 1 to 'arrayC' positions which correspond to 'arrayA' elements.*
__global__ void	**gpu_sumC**(int *arrayC, int len) *This is the kernel that sums every 'arrayC' element with the previous one.*
__global__ void	**gpu_lastKernel**(int *arrayA, int *arrayC, int *sorted, int n) *This is the kernel that sorts 'arrayA' using 'arrayC' and puts the result in 'sorted'.*

Public functions

Type	Name
float	**initArray**(int *array_h, int n, int range, int blockSize) *This is the function that creates and initializes a random array, calling the appropriate kernel, and puts it in 'array_h'.*
float	**countingSortDEVICE**(int *array_h, int n, int max, int blockSize) *This is the function that sorts 'array_h' using Counting Sort algorithm on the GPU.*
void	**countingSortHOST**(int *array, int n, int max) *This is the function that sorts 'array' using Counting Sort algorithm on the CPU.*
void	**make_csv**(int blockSize, float elapsedInit, float elapsedSort, int n, int range) *This is the function that creates a file ".csv" which contains values for 'blockSize', 'gridSize', 'elapsedInit', 'elapsedSort'.*

Kernels documentation

kernel gpu_initArray

```
__global__ void gpu_initArray(
        int *arrayA,
        int n,
        int range,
        int seed
)
```

Parameters:

- arrayA pointer to the unsorted array.
- n number of array elements.
- range maximum acceptable integer.
- seed seed of random number.

kernel gpu_fullC

```
__global__ void gpu_fullC(
        int *arrayA,
        int *arrayC,
        int n
)
```

Parameters:

- arrayA pointer to the unsorted array.
- arrayC pointer to the auxiliary array.
- n number of array elements.

kernel gpu_sumC

```
__global__ void gpu_sumC(
        int *arrayC,
        int len
)
```

Parameters:

- arrayC pointer to the auxiliary array.
- n number of array elements.

kernel gpu_lastKernel

```
__global__ void gpu_lastKernel(
        int *arrayA,
        int *arrayC,
        int *sorted,
        int n
)
```

Parameters:

- arrayA pointer to the unsorted array.
- arrayC pointer to the auxiliary array.
- sorted pointer to the sorted array.
- n number of array elements.

Functions documentation

function initArray

```
float initArray(
        int *array_h,
        int n,
        int range,
        int blockSize
)
```

Parameters:

- array_h pointer to the unsorted array.
- n number of array elements.
- range maximum acceptable integer.
- blockSize number of threads in each block.

function countingSortDEVICE

```
float countingSortDEVICE(
        int *array_h,
        int n,
        int max,
        int blockSize
)
```

Parameters:

- array_h pointer to the unsorted array.
- n number of array elements.
- max maximum acceptable integer.
- blockSize number of threads in each block.

function countingSortHOST

```
void countingSortHOST(
        int *array,
        int n,
        int max
)
```

Parameters:

- array — pointer to the unsorted array.
- n — number of array elements.
- max — maximum acceptable integer.

function make_csv

```
void make_csv(
        int blockSize,
        float elapsedInit,
        float elapsedSort,
        int n,
        int range
)
```

Parameters:

- blockSize — number of threads in each block.
- elapsedInit — time to initialize the array.
- elapsedSort — time to sort the array.
- n — number of array elements.
- range — maximum acceptable integer.

How to run

To generate measures and obtain further information you can use this Google Colab notebook.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.