# High Performance Computing

Counting Sort with CUDA

A.Y. 2021/22

Version 1.0.0


| | | |
|---|---|---|
| Alessio Pepe | 0622701463 | a.pepe108@studenti.unisa.it |
| Teresa Tortorella | 0622701507 | t.tortorella3@studenti.unisa.it |
| Paolo Mansi | 0622701542 | p.mansi5@studenti.unisa.it |


# 1 Index

# 2 Problem Description

## 2.1 Assignment

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

## 2.2 Algorithm description

Counting sort is a sorting algorithm that sorts the elements of an array of integer value by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

The meta-algorithm of the counting sort is the following (from https://it.wikipedia.org/wiki/Counting_sort):

```
countingSort(fileName)
   A[] = read(fileName)

   //Calcolo degli elementi max e min
   max ← A[0]
   min ← A[0]
   for i ← 1 to length[A] do
      if (A[i] > max) then
         max ← A[i]
      else if(A[i] < min) then
         min ← A[i]
   end for

   * creates a C array of size max - min + 1
   for i ← 0 to length[C] do
      C[i] ← 0
   end for
   for i ← 0 to length[A] do
      C[A[i] - min] = C[A[i] - min] + 1
   end for
   k ← 0
   for i ← 0 to length[C] do
      while C[i] > 0 do
         A[k] ← i + min
         k ← k + 1
         C[i] ← C[i] - 1
   end for

   write(fileName, A[])
```

The algorithm consists of 3 parts:

- find the maximum and minimum value within the array to calculate the width of the range of values.
- use an auxiliary array to counter the frequency of each single value.
- repopulate the array in an orderly manner based on the newly calculated frequency array.

The computational complexity of the algorithm is given by the sum of the computational complexities of the three parts and is in total

$$\big(O(n)\big) + \big(O(n) + O(k)\big) + \big(O(n)\big)$$

where:

- n is the number of elements of the array.
- k is the size of the range of elements in the array (we are considering on average that the allocation of a block of memory costs $O(1)$.

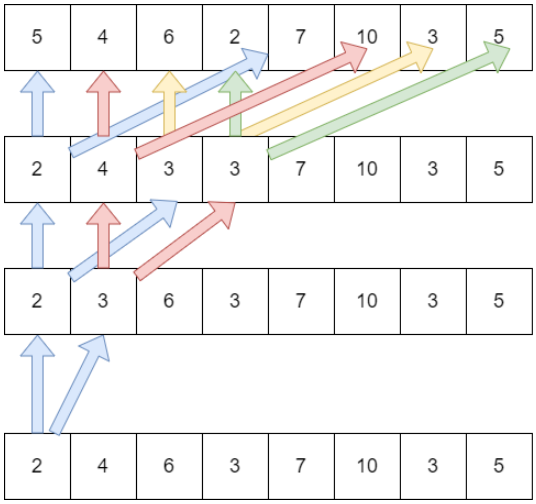
We are aware that it is not possible to parallelize the PMF vector allocation operation and therefore the theoretical speedup curve should not be that of the images that will be shown in the solutions section later,

but since this algorithm makes sense to be used when $k \ll n$ then we consider that represented a good approximation.

## 2.3 Solution idea

### 2.3.1 Min-max

Two different kernels are used that perform the same operation. however, one of these works for subsequent reductions. The kernel works as in the image.

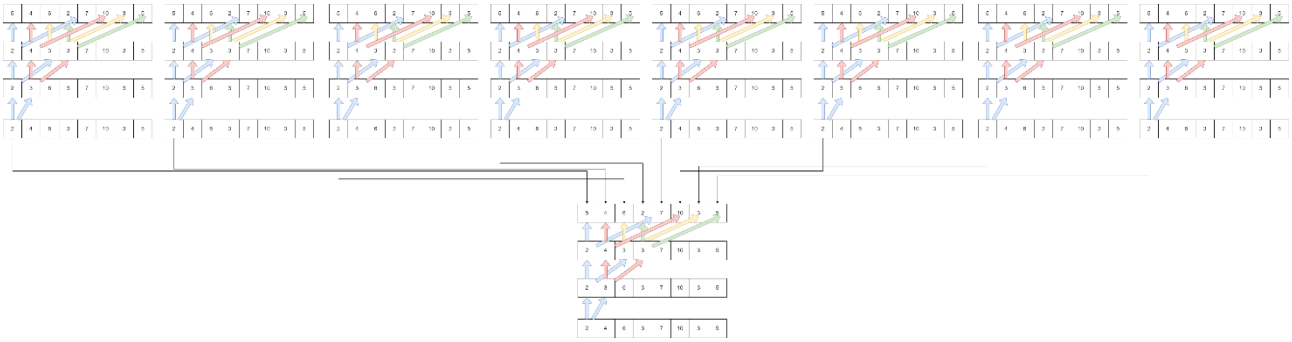



The threads of each block copy their part of the array inside the shared memory and calculate the minimum and maximum in

$$O(log(n))$$

iterations, where n is the size of the block.

Being that a minimum and a maximum are calculated for each block, the kernel is launched which calculates the minimum and maximum among those found before up to the global ones. What is described is shown in figure.



Here the use of global memory was not taken into consideration because there would have been problems as a large number of reads and writes are carried out. Not even texture memory as it is read only.

## 2.4 Counting occurrence

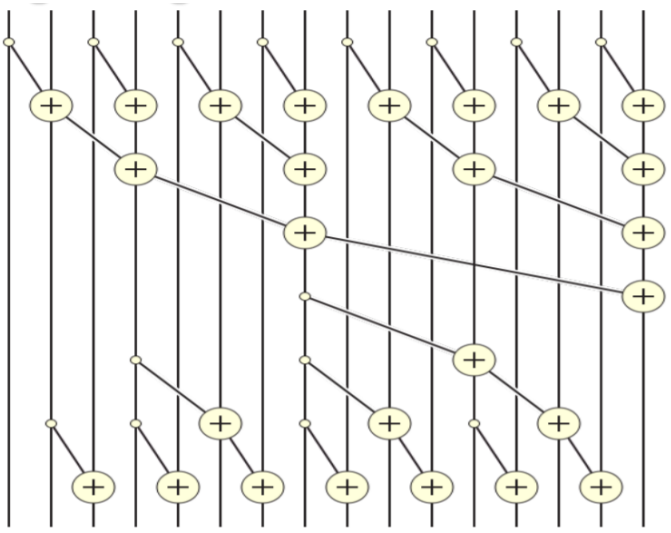
We have set up a kernel to count occurrences where threads cover the array, each block computes a local occurrence vector kept in shared memory and at the end each block increments the global vector.

We used shared memory as there are a lot of write and read operations (which must also be atomic). The fact that so many local vectors are used results in fewer conflicts in atomic scripts.

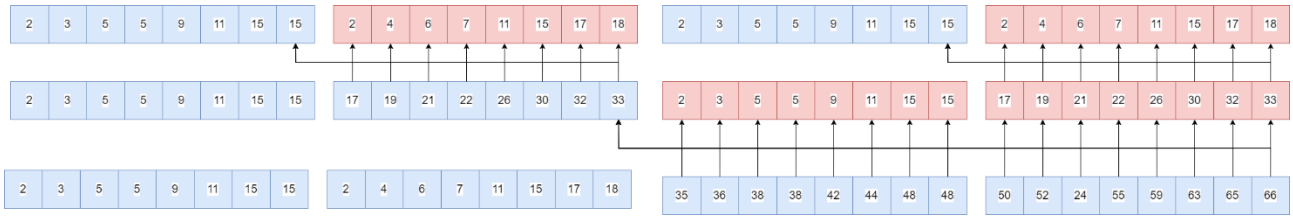
The use of global memory has not been taken into consideration as given the number of read and write operations the time would have been significantly longer. Not even texture memory as it is read only.

### 2.4.1 Populating array – CDF division

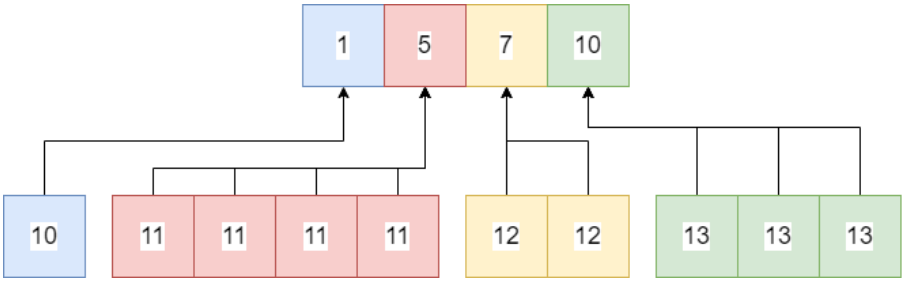
To populate the array, we first computed the CDF from the PMF. To do this we have implemented a kernel that performs the scan algorithm efficiently in parallel, using a logarithmic number of iterations as shown in figure.



After that, an additional kernel to be able to transform pieces of local CDF into a global CDF as shown in figure.




To perform the population, we used a kernel where the threads cover the CDF vector. Each thread calculates the value to insert into the array and the locations to insert it and populates that part of the array as shown in figure.



### 2.4.2 Total solution

We have provided two files countingsort.cu and countingsort_mhost.cu (which contain the described version and the same version using host CUDA malloc based on the evaluation of the results).

We have also provided a Colab notebook which is useful for compiling and running the application using the Colab GPU. Furthermore, we have provided an excel file with preset formulas to copy and paste the measurement values obtained with the notebook to evaluate performance.

# 3 How to run measure

In the Colab notebook supplied it is necessary to sequentially execute the blocks present. These contain:

- the installation of the libraries.
- The compilation of the application.
- The generation of the measures.

The measurements are unfortunately printed in the output due to problems of RAM memory occupation and file system synchronization with Drive. By copying and pasting the output of the cells in the excel file provided at the positions indicated in the notebook before the cell, all the graphs and other information are automatically generated.

# 4 Experimental Setup

## 4.1 OS setup

Colab is used.

## 4.2 GPU

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Mon_Oct_12_20:09:46_PDT_2020
Cuda compilation tools, release 11.1, V11.1.105
Build cuda_11.1.TC455_06.29190527_0
Wed Jan 19 10:30:33 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 495.46       Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla P100-PCIE...  Off  | 00000000:00:04.0 Off |                    0 |
| N/A   31C    P0    25W / 250W |      0MiB / 16280MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
```

## 4.3 CPU

```
processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 79
model name      : Intel(R) Xeon(R) CPU @ 2.20GHz
stepping        : 0
microcode       : 0x1
cpu MHz         : 2199.998
cache size      : 56320 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 3dnowprefetch invpcid_single ssbd ibrs
ibpb stibp fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm rdseed adx smap xsaveopt
arat md_clear arch_capabilities
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs taa
bogomips        : 4399.99
clflush size    : 64
cache_alignment       : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

## 4.4 Memory

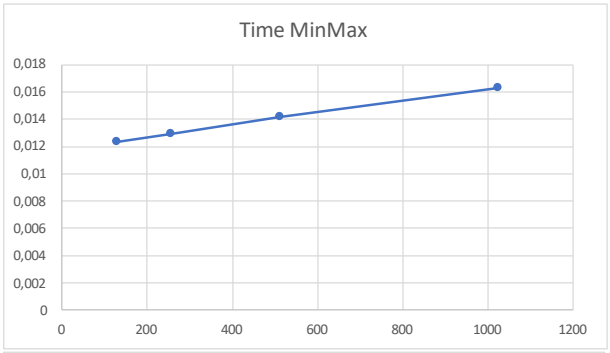
```
MemTotal:       13302920 kB
MemFree:         8475604 kB
MemAvailable:   12374508 kB
Buffers:           71052 kB
Cached:          3915844 kB
SwapCached:            0 kB
```

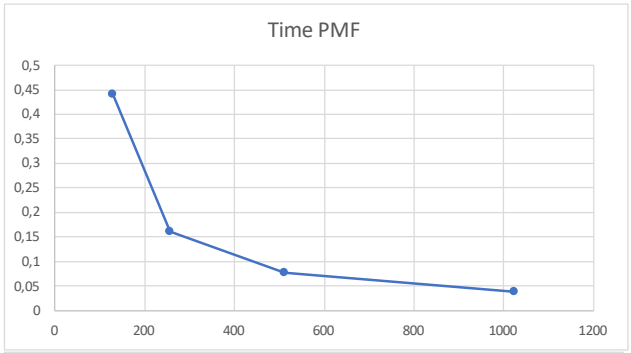
```
Active:            740556 kB
Inactive:         3718028 kB
Active(anon):      455956 kB
Inactive(anon):       500 kB
Active(file):      284600 kB
Inactive(file):   3717528 kB
Unevictable:            0 kB
Mlocked:                0 kB
SwapTotal:              0 kB
SwapFree:               0 kB
Dirty:                916 kB
Writeback:              0 kB
AnonPages:         471668 kB
Mapped:            162980 kB
Shmem:               1200 kB
KReclaimable:      222724 kB
Slab:              273192 kB
SReclaimable:      222724 kB
SUnreclaim:         50468 kB
KernelStack:         5872 kB
PageTables:          6924 kB
NFS_Unstable:           0 kB
Bounce:                 0 kB
WritebackTmp:           0 kB
CommitLimit:     6651460 kB
Committed_AS:    3660072 kB
VmallocTotal:   34359738367 kB
VmallocUsed:        46704 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:       451392 kB
DirectMap2M:     11079680 kB
DirectMap1G:      4194304 kB
```

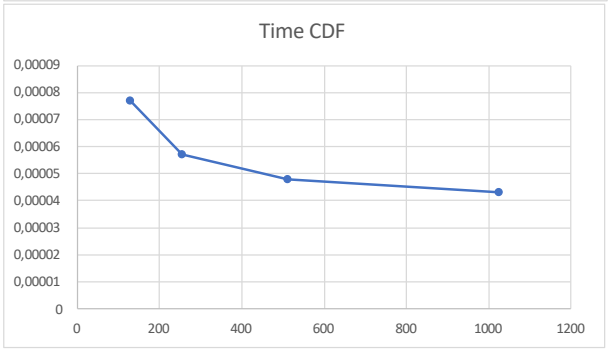
# 5 Results

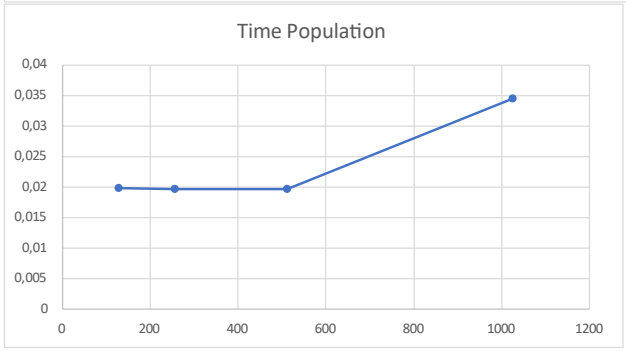
## 5.1 Block size valutation

Four block sizes (128, 256, 512, 1024) have been evaluated for the four kernels that allow the single SM to have an occupancy of 100% to study its performance. In the graphs below we can see the result of this study.




The best dimension for minmax is 128. This is because in this way the occupancy in the subsequent reduction operations will be higher.

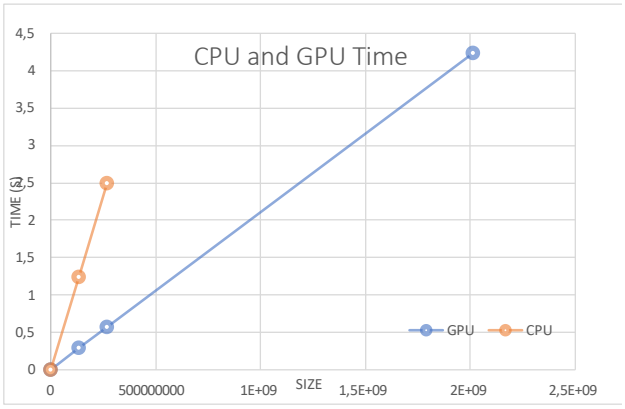
As for the PMF, the best size is 1024 (the largest possible). This is because in this way fewer write operations must be performed inside the global memory.

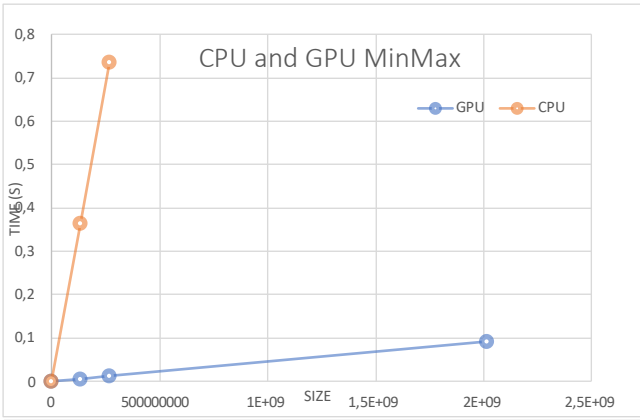
As far as the CDF is concerned, the best size is 1024. This is because the scan algorithm used is more efficient than the reduction algorithm.

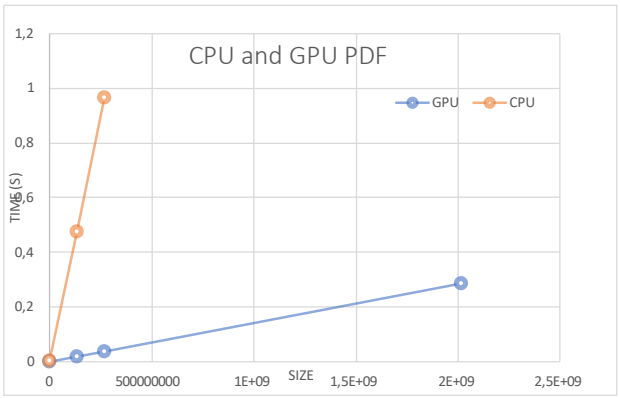
As for the population, the best block size is 512, probably related to the warp size.

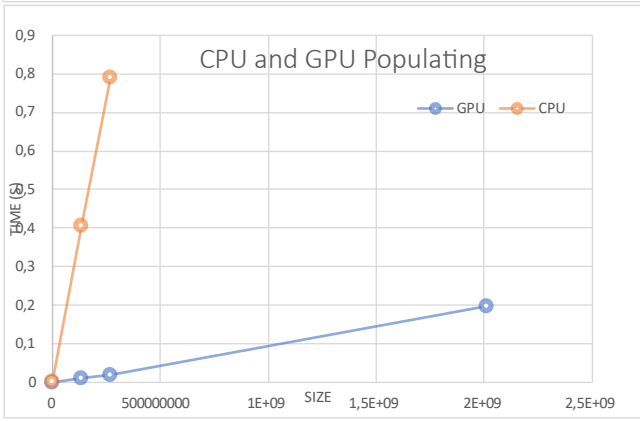
## 5.2 Time valutation

We evaluated the algorithm in its entirety and its sub-parts and compared them with the results obtained on a single CPU (obtained using the code provided for the OpenMp contest).

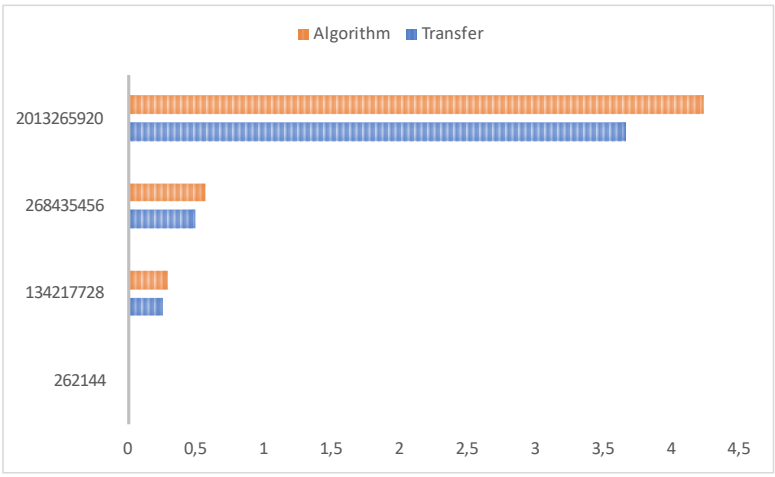


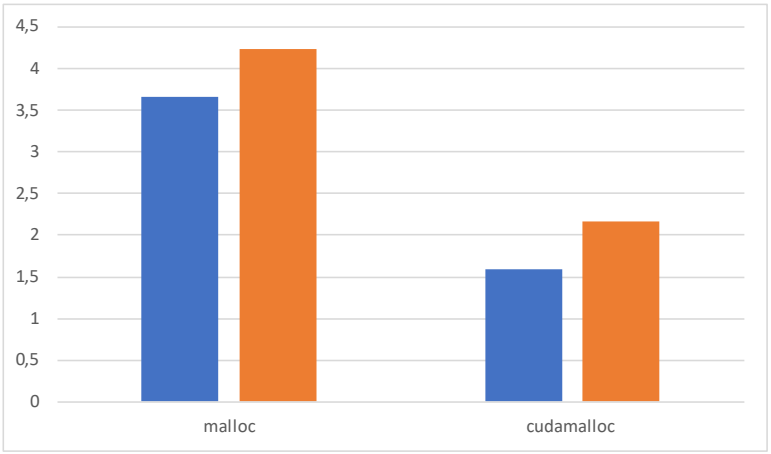




We can see that compared to the three sub-algorithms it has a notable increase in performance. On the other hand, it seems that in the total algorithm this is less evident.




Therefore, evaluating the time of the transfers compared to that of the execution of the rest of the code, we note that the real bottleneck is the transfer. This is the reason why we have provided the second version. This second version plans to use CUDA Malloc hosts instead of the operating system malloc. This is to allocate the array directly in the Pinned Memory avoiding the transfer from the Pageable Memory to the Pinned Memory.




We note that the transfer time is halved and with it also the execution time of the algorithm.

A further improvement could be obtained by parallelizing the execution using small parts of arrays on each stream.

## 5.3 GFlop Evalutation

Using the NVProf profiler we obtained the number of average operations performed by the kernels for the various sizes, and we used them to evaluate the number of Floating-Point operations per second as shown in the table.

| size | range | GFmm | GFpdf | GFcdf | GFpp |
|---|---|---|---|---|---|
| 262144 | 12288 | 464,9225 | 650,9842 | 116,8557 | 36,80915 |
| 134217728 | 12288 | 1211,916 | 1187,825 | 105,9009 | 71,72731 |
| 268435456 | 12288 | 1217,487 | 1191,808 | 101,0009 | 69,57148 |
| 2013265920 | 12288 | 1220,569 | 1193,022 | 88,78797 | 51,13602 |

# License