Politecnico di Torino
GPU Programming

REPORT: PROJECT
WEBCAM STREAMING & FILTERS

January 13, 2022

Matteo Battilana, Salvatore Gabriele La Greca, Giovanni Pollo

Contents

1 Introduction

A classical video streaming algorithm over internet or over any other communication channel is based on the concept of sending not frame by frame as they are but instead they are based on sending the first frame and then the difference between the new one and the previous, where difference here is to be indended pixel-by-pixel difference.

If this can be arguable for video with a lower resolution, the heaviness of sending each frame as it is for high video resolution is notable. Let's take as an example a FULL HD video, means that each frame is composed of 1920x1080 pixels where 1920 is the width of the frame while 1080 is the height. Supposing the frame is in RGB24 format, means that each pixel is rapresented by 3 byte (one for each channel R, G, and B).

By doing a rapid computation, each frame measures $3 \cdot 1920 \cdot 1080B = 6220800B = 5.93MB$. Supposing now the video used by example is a 30 fps video, means that each second we have 30 frame, each of one measures 5.93 MB: each second we are sending about 178 MB. To send 178 MB/s we need a transfer link bandwith of 1492 Mbps, that is unfeasible.

So the solution is to send the difference, and this means sending only the pixels that change or, better, pixels where their difference is above a certain threshold.

The purpose of the project is to demonstrate the performances that are obtainable by computing the difference on a CPU and to compare them with the ones obtainable by using a General Purpose GPU or *GPGPU*. A Nvidia GPU will be used for the benchmark and therefore the code will be based on CUDA. Among all these considerations, different filters will be added in order to demonstrate the potentiality of GPGPU computations on video elaboration and streaming.

2 Video Streaming

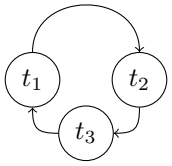
Before looking at the algorithm that computes the difference between frames, is important to give a look at how frames are captured and the overall architecture of the software. It's a client server architecture where the server transmits the difference to the client via a socket.

All the different variants of implementation that will be analyzed produces three outputs:

- `h_pos`: the number of pixels that are different and greather than a certain threshold.
- `h_diff`: in code it is actually the array of the current frame that after the application of the algorithm contains the difference of each byte.
- `h_xs`: it's a mapping vector for the `h_diff`. This means that the `h_diff[0]` is the difference of byte elemnt at position `h_xs[0]`.

In order to capture frame by frame from the webcam and to visualize them, OpenCV is used. It's not so efficient in terms of performances, especially on the platform used (Nvidia Jetson Nano with 4 ARM cores @ 1.5 GHz) but for the purpose of this project it will be fine. The most important thing to underline is that a frame will be rapresented by the OpenCV's object `Mat` that contains, among other informations, the dimension of the image (that is fixed to a FULL HD resolution) and an array of `uint8_t` items, each representing a channel of a pixel for each pixels of the image. The array can be allocated automatically at the creation of the `Mat` object or an external array can be used and later on this feature will be exploited.

The aim is to have an efficient software, so a multi thread approach is adopted. There are, in fact, 3 different threads each of them with a different purpose: capture, elaborate and send. They work in a circular way. Means that the capture thread is a producer for the elaborate one, the elaborate one is a producer for the send one and the last one is a producer for the capture one. In this way we have all the threads working at the same time on a different task. The t2 thread is where the magic happens so where the elaboration of the difference is executed.



In the following pages, for metric considerations these terms will be used:

- `fps`: number of frames per second.
- `read`: time of execution of the capture thread.
- `for`: time of execution of the elaboration thread. This is what is under discussion in this project.

2.1 CPU Implementation

The CPU implementation is the easier one and the most basic implementation of the algorithm. It consists on a loop between each byte that compose the two frames (the current one and the previous one) and compute the difference, storing it in a new vector.

The C++ implementation is the following:

```
1  int total = 3 * ctx.sampleMat->rows * ctx.sampleMat->cols; // no. bytes
2  Mat pvs = pready->pframe->clone();
3  pready->h_pos = 0;
4
5  for (int i = 0; i < total; i++) {
6    int df = pready->pframe->data[i] - previous.data[i];
7    if (df < -LR_THRESHOLDS || df > LR_THRESHOLDS) {
8      pready->pframe->data[pready->h_pos] = df;
9      pready->h_xs[pready->h_pos] = i;
10     pready->h_pos++;
11   } else {
12     pvs.data[i] -= df;
13   }
14 }
15
16 previous = pvs;
17
```

The code here is pretty simple. For each byte, the difference `df` is computed. If this difference is greater than a fixed threshold `LR_THRESHOLDS` means that it is a valid difference so it can be sent. For all the benchmarks proposed in the following, the value of `LR_THRESHOLDS` is fixed to 20. The result of the reconstructed frame at the client side can be seen at Figure 1.

Figure 1: *The result of the reconstructed frame*

The important point here is that if the difference is too low, it can't be simply discarded, so a kind of negative feedback is needed. This is the purpose of the line of code at line 7, where the value of the byte of the current frame (that at the end of the loop will be the previous for the next iteration) is itself minus the value of the difference. This means that at the next iteration, if that value changes again and its difference increases it will take under consideration as a big difference. Without this, there will be a sum of errors in the reconstructed image, leading in a wrong visualization. The result if the error is not considered can be seen at Figure 2.

Figure 2: *What happens after a certain time if the error is not take under consideration*

The performances here are pretty bad. By means of in-code time measurements, the video streaming is stable at 7 fps with an average `for` time of 140.0 ms and an average `read` time of 0.0 ms. This means that here the big bottleneck is due to the elaboration part of the software.

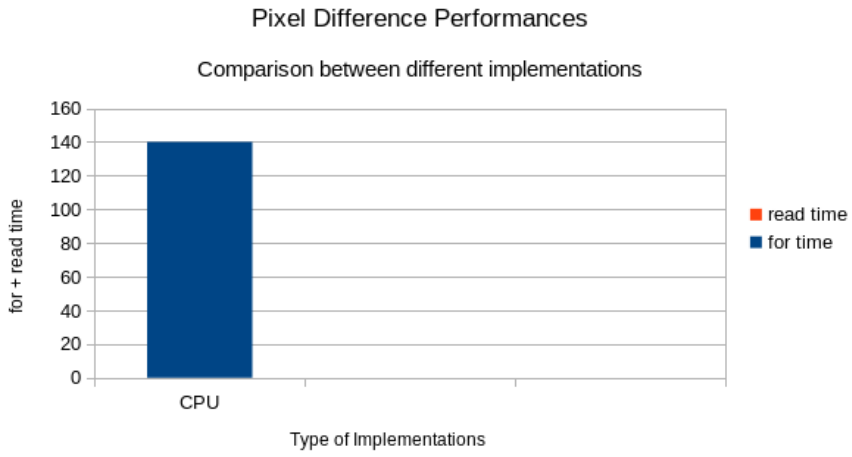


Figure 3: *Pixel Difference Comparison*

2.2 GPU Implementation

The idea is now to port this in the GPU. The GPU can be seen as an accelerator for the CPU, an accelerator that is capable of a high degree of parallelism by executing a lot of simple threads all in parallel in a SIMD way.

In order to port this in CUDA and execute the computation on the GPU, a series of considerations needs to be done. First of all, it's not said that by executing this in GPU there is a direct improvement in performance. This is due to the fact that there is a big bottleneck between the CPU and the GPU and this is the bus that connects them. A GPU in order to do some work

on some data needs to have them in its memory (so the GPU can't access directly the CPU's RAM).

The first thing to do, before defining the kernel (the piece of code that is offloaded to the GPU), is to understand how split the 6220800 bytes among the concurrent threads of the GPU. In the Mat object, the frame is rapresented row-major, means that rows are at consecutive address, as shown in Table 1.

0	1	2
3	4	5
6	7	8

0	1	2	3	4	5	6	7	8
0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8

Table 1: *Row major rapresentation in memory. On top the value, on bottom the address*

Threads can be organized differently according how they access the memory, and in this case there can be two possible cases of memory access per thread:

Thread 0	0	3	6
Thread 1	1	4	7
Thread 2	2	5	8

Table 2: Non consecutive access

Thread 0	0	1	2
Thread 1	3	4	5
Thread 2	6	7	8

Table 3: Consecutive access

So the entire array can be divided into <N> chunks where <N> is the number of threads to launch on the GPU. By organizing the kernels in a way they can access the memory in a consecutive access, the kernel will achieve the so called *memory coalesced access*.

How many threads? For a first implementation, and in order to have a modular implementation of the code (so it can be executed on different GPUs with different capabilities), the following code is adopted:

```
struct cudaDeviceProp prop;
cudaGetDeviceProperties(&prop, 0); // retrieves device infos

int total = 3 * ctx.sampleMat->rows * ctx.sampleMat->cols; // no. bytes
int nMaxThreads = prop.maxThreadsPerBlock;
int maxAtTime = total / nMaxThreads;

```

According to this, each thread will work on $N = maxAtTime$ consecutive bytes, dividing the memory into *nMaxThreads* chunks, and overlapping is totally avoided. The GPU used for the experiments allows a maximum of 1024 threads to be executed concurrently.

For what concerns the memory allocation, obviously the CPU address space and the GPU one are two separated things, so a specific allocation on the device side must be done. This can be accomplished thanks to some CUDA's API as follows:

```
uint8_t *d_current, *d_previous, *d_diff;
int *d_xs;
unsigned int *d_pos;

cudaMalloc((void **)&d_diff, total * sizeof *d_diff);
cudaMalloc((void **)&d_xs, total * sizeof *d_xs);
cudaMalloc((void **)&d_current, total * sizeof *d_current);
cudaMalloc((void **)&d_previous, total * sizeof *d_previous);
cudaMalloc((void **)&d_pos, sizeof *d_pos);

```

2.2.1 Naif version

The naif version of the implementation is to port as it is the CPU code into a kernel. From the host side (the CPU) that asks to the device (the GPU) to execute the kernel, there are three kind of operations to be done:

1. copy the frame into the GPU's memory
2. launch the kernel
3. copy back the results from the GPU's memory to the HOST memory

In the naif implementation, both the previous and current frames are copied into the GPU's memory. At the end of the execution, the results are copied back. This implies 2 transfers HostToDevice (HtoD) for current and previous and 3 transfers DeviceToHost (DtoH) for h_xs, h_diff and h_pos. A memset on the device side is required too to set the initial value of d_pos to 0.

The host will therefore execute the following operations:

```
cudaMemset(d_pos, 0, sizeof *d_pos);

// launch
cudaMemcpy(d_previous, previous.data, total, cudaMemcpyHostToDevice);
cudaMemcpy(d_current,pready->pframe->data,total,cudaMemcpyHostToDevice);
kernel<<<1,nMaxThreads>>>(d_current,d_previous,d_diff,maxAtTime,d_pos,d_xs);

// copy back
cudaMemcpy(pready->pframe->data, d_diff, total, cudaMemcpyDeviceToHost);
cudaMemcpy(pready->h_xs, d_xs, total*sizeof *d_xs, cudaMemcpyDeviceToHost);
cudaMemcpy(&pready->h_pos, d_pos, sizeof *d_pos, cudaMemcpyDeviceToHost);

// copy previous
cudaMemcpy(previous.data, d_current, total, cudaMemcpyDeviceToHost);

```

It's possible to notice a pretty similar structure of the CPU implementation, where the *for loop* is replaced by the kernel. The kernel is launched with a grid dimension of (1, 0, 0) and a block dimension of (*nMaxThreads*, 0, 0), thus only the x dimension is used because the kernel is coded in such a way that the chunk assigned to it is analyzed in a linear way.

In order to define the code of the kernel, there are two main problems to be analyzed first:

1. How does the kernel knows at which address it must work?
2. How is possible to fill a variable size array in a concurrent environment in CUDA?

The first problem is easily solved, it's only a matter of indexes. Each kernel needs to know which chunk of the memory is assigned to it so first of all it computes it's own unique index that will space from 0 to $nMaxThreads - 1$. Once this is defined, the kernel knows that it must work on the i-th chunk but this needs to be translated into the real index to be used to address the memory. Because each size is *maxAtTime* large, means that the the real index that identifies the start of the chunk is $threadIndex \cdot maxAtTime$.

The second problem is a bit more complicated to solve. In the CPU implementation a variable h_pos is used in order to indicize the resulting vectors an incremental way and at the end this variable is used to know the length of the result. The problem when this concept is trasposed to the GPU world is that there is a global variable d_pos that is used by *nMaxThreads* concurrently so there is the need to read and increment this variable in a safe way, so only a kernel at a time can ready and modify it. Luckly, the CUDA framework offers a series of functions that ensures atomic operations between kernels, and in this case the needed one is `atomicInc(unsigned int *address, unsigned int val)` where *address* is the pointer to the value that needs to be atomic incremented, *val* is the wraparound value, so the max value that the variable can reach before going again to 0, and it returns as result the value atomically incremented.

After solved those problems, a look at the real implementation of the kernel can be given:

```
__global__ void kernel(
uint8_t *current, uint8_t *previous, uint8_t *diff,
int maxSect, unsigned int *pos, int *xs) {
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  unsigned int npos;
  int df;

```

```
8
9     int start = x * maxSect;
10    int max = start + maxSect;
11
12    #pragma unroll
13    for (int i = start; i < max; i++) {
14
15      df = current[i] - previous[i];
16      if (df < -LR_THRESHOLDS || df > LR_THRESHOLDS) {
17        npos = atomicInc(pos, 6220801);
18        diff[npos] = df;
19        xs[npos] = i;
20      } else {
21        current[i] -= df;
22      }
23
24    }
25
26  }
27
```

As can be seen, apart the computation of the start and end index of the computation, the for loop is pretty much the same as the CPU implementation. The only difference is the one concerting the use of the `atomicInc` function as explained before.

The performances here are slightly better than the CPU implementation. The video streaming is elaborated at around 13 fps with an average `for` time of 70.0 ms and an average `read` time of 0.0 ms, and the comparison with the previous case can be seen in Figure 4.

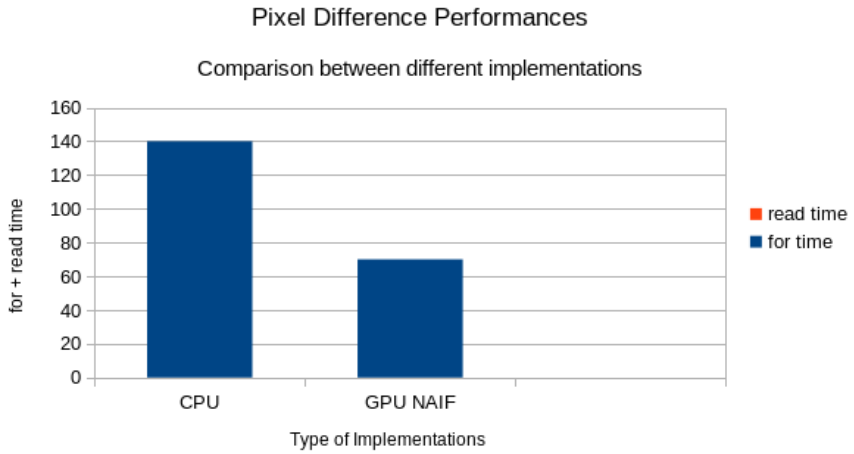


Figure 4: *Pixel Difference Comparison*

This means that the big bottleneck is still due to the elaboration part of the software. Using the profiler offered by Nvidia *nvprof* executed for 30 seconds, it's possible to get a lot more details about the time required by each GPU operation, and in particular what concerns this project is the transfer time and the kernel execution time. The results are shows in table 4.

Operation	Avg (ms)	Min (ms)	Max (ms)
Kernel	40.295	35.912	47.033
DtoH	4.2975	1.5100	52.878
HtoD	3.3280	2.4544	25.808
Memset	0.0008	0.0007	0.0017

Table 4: Profiling results

2.2.2 Asynchronous APIs and removal of useless operations

In the previous implementation, a big portion of the execution time is covered by the kernel execution but the two data transfers are not negligible. The total average time is about 46 ms means that the *for* time is for the 60% due to the GPU computation and the remaining part is related to host taks.

CUDA offsers a variant of the APIs that are asynchronous with respect to the device execution. This means that when one of these APIs is called, the operation starts on the device side but the control returns immediately to the host despite the end of the operation on the GPU.

This means that is possible to give to the device a series of operations to execute and meanwhile the host can work on its own operations. In this specific case the host doesn't have any task to complete so a function `cudaDeviceSynchronize()` is used to wait for the device to complete its operations. Howewer this solution is slightly more efficient because is possible to assign to the GPU all the jobs in a consecutive way then waiting instead of assign a job at a time.

```
cudaMemsetAsync(d_pos, 0, sizeof *d_pos);

cudaMemcpyAsync(d_previous, previous.data, total, cudaMemcpyHostToDevice);
cudaMemcpyAsync(d_current,pready->pframe->data,total,cudaMemcpyHostToDev);
kernel<<<1,nMaxThreads>>>(d_current,d_previous,d_diff,maxAtTime,d_pos,d_xs);

cudaMemcpyAsync(pready->pframe->data, d_diff, total,cudaMemcpyDeviceToHost);
cudaMemcpyAsync(pready->h_xs,d_xs,total*sizeof *d_xs,cudaMemcpyDevToHost);
cudaMemcpyAsync(&pready->h_pos,d_pos,sizeof *d_pos, cudaMemcpyDeviceToHost);

cudaMemcpyAsync(previous.data, d_current, total, cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();

```

Another optimization that can be done is to remove a particular useless copy both HtoD and DtoH, and that is the one regarding the previous frame. In fact, after the first launch of the kernel, the previous frame is already stored in the GPU's memory so it's avoidable the back copy to the host and gain to the device. What can be done is to implement a simple pointer swap, in fact after the first execution the current frame will become the previous frame at the next round. So there is a pool of two pointers that alternatively works one as the current and the other one as the previous.

The host code becomes like this:

```
// current-previous swap
uint8_t *d_prev = d_current;
d_current = d_previous;
d_previous = d_current;

cudaMemsetAsync(d_pos, 0, sizeof *d_pos);

// Copy in the current pointer and run
cudaMemcpyAsync(d_current,pready->pframe->data,total,cudaMemcpyHostToDeve);
kernel<<<1,nMaxThreads>>>(d_current,d_previous,d_diff,maxAtTime,d_pos,d_xs);

cudaMemcpyAsync(pready->pframe->data,d_diff,total,cudaMemcpyDeviceToHost);
cudaMemcpyAsync(pready->h_xs,d_xs,total*sizeof *d_xs,cudaMemcpyDevToHost);
cudaMemcpyAsync(&pready->h_pos,d_pos,sizeof *d_pos,cudaMemcpyDeviceToHost);

cudaDeviceSynchronize();

```

Another optimization in terms of copies regards the download of the two arrays forming the result. In fact, instead of copying all the *total* bytes is smarter to cpy only *d_pos* bytes, the ones that are really useful.

So the first download copy regards *d_pos*, then a `cudaDeviceSynchronize()` is executed in order to wait for the complete of all the previous operation and getting a valid *d_pos*, then the remaining data are copied.

The host code becomes:

```
1  // current-previous swap
2  uint8_t *d_prev = d_current;
3  d_current = d_previous;
4  d_previous = d_current;
5
6  cudaMemsetAsync(d_pos, 0, sizeof *d_pos);
7
8  // Copy in the current pointer and run
9  cudaMemcpyAsync(d_current,pready->pframe->data,total,cudaMemcpyHostToDeve);
10 kernel<<<1,nMaxThreads>>>(d_current,d_previous,d_diff,maxAtTime,d_pos,d_xs);
11
12 cudaMemcpyAsync(&pready->h_pos,d_pos,sizeof *d_pos,cudaMemcpyDeviceToHost);
13 cudaDeviceSynchronize();
14
15 cudaMemcpyAsync(pready->pframe->data,pready->h_pos,total,cudaMemcpyDToHt);
16 cudaMemcpyAsync(pready->h_xs,d_xs,pready->h_pos*sizeof *d_xs,cdMemcpyDToH);
17
18
19 cudaDeviceSynchronize();
20
```

Last but not least, the *cudaMemsetAsync* at line 6 can be removed completely. In fact, it's possible to move the initial reset of the *d_pos* counter by doing it at the beginning of the kernel and it's easily done even in a concurrent environment by letting only one kernel to perform the reset. This can be achieved by putting a condition on the kernel index. In this case it will be done by the first kernel. In order to have all the kernel executing more or less in parallel the same instructions, after the reset of the counter, a `__syncthreads()` is executed that assures that all threads are synchred untill that point.

Thus, the kernel code becomes:

```
1  __global__ void kernel(
2  uint8_t *current, uint8_t *previous, uint8_t *diff,
3  int maxSect, unsigned int *pos, int *xs) {
4    int x = threadIdx.x + blockDim.x * blockIdx.x;
5    unsigned int npos;
6    int df;
7
8    // counter reset done by thread with index = 0
9    if (!x) *pos = 0;
10   __syncthreads();
11
12
13   int start = x * maxSect;
14   int max = start + maxSect;
15
16   #pragma unroll
17   for (int i = start; i < max; i++) {
18
19     df = current[i] - previous[i];
20     if (df < -LR_THRESHOLDS || df > LR_THRESHOLDS) {
21       npos = atomicInc(pos, 6220801);
22       diff[npos] = df;
23       xs[npos] = i;
24     } else {
25       current[i] -= df;
26     }
27
28   }
29
30 }
31
```

All these optimizations let achieve a big result in performance terms. The video streaming is elaborated at around 20 fps with an average `for` time of 55.0 ms and an average `read` time of 0.0 ms, and the comparison with the previous case can be seen in Figure 5.

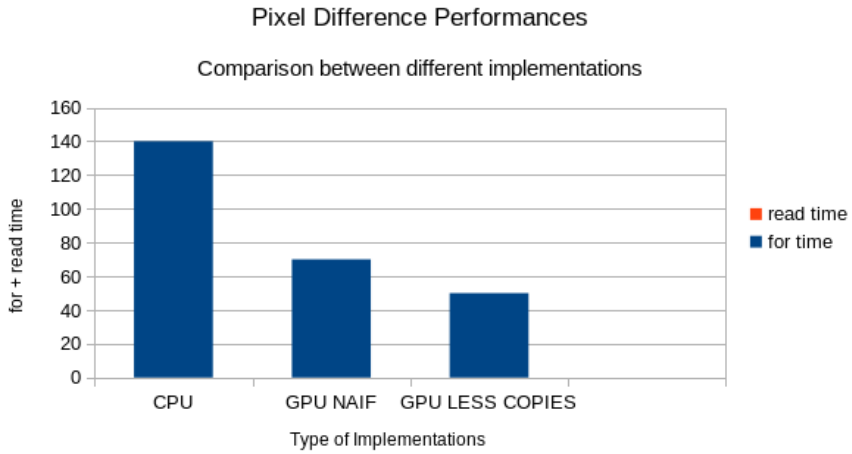


Figure 5: *Pixel Difference Comparison*

The results of the *nvprof* profiler are shown in Table 5 and is noticeable the big improvement in terms of performances for what concerns the DtoH transfer. The HtoD transfer remains more or less the same but it's executed only one time instead of two as the naif implementation.

Operation	Avg (ms)	Min (ms)	Max (ms)
Kernel	41.426	37.278	46.423
DtoH	0.0227	0.0009	9.2925
HtoD	3.7813	2.1733	14.188

Table 5: Profiling results

2.2.3 Pinned or page-locked memory

Untill now a good bottleneck has been removed and it is the transfer from the Device to the Host. What about the Host to Device transfer? In order to solve this issue an understanding on how memory is managed by the CPU and how the GPU transfers data from the CPU's managed memory and the GPU.

What a modern operating system does is to manage the physical memory through the concept of pages and thus the virtual memory. The address space of the virtual memory is different from the physical one, thus a translation is needed. In order to let the hardware translate from a space to another, a special hardware called translation lookaside buffer (TLB) is used. It's a memory cache that is used to reduce the time taken to access a user memory location and it's part of the chip's memory-management unit (MMU) needed in fact by all the modern operating systems. The TLB stores the recent translations of virtual memory to physical memory and can be called an address-translation cache.

When a CUDA's memcpy is invoked, the DMA starts transfering data but as soon as the translation is not anymore stored in the cache, the operating system has to solve this situation by loading a new page. Thus this is not really a asynchronous operation and it's highly cpu-dependent. Thus the velocity of the copy is based on the speed of the CPU and in edge devices where the CPU is not so performant as a desktop CPU the overhead of this mechanism can't be neglected.

The solution is to allocate a piece of memory that is pinned or *page locked*. Executing an allocation like this is like telling the operating system virtual memory manager that the memory pages must stay in physical ram so that they can be directly accessed by the GPU across the PCI-express bus. And it's a lot faster thanks to the DMA: when the memory is page locked, the GPU DMA engine can directly run the transfer without requiring the host CPU, which reduces overall latency and decreases transfer times.

CUDA offers an API to do this, called `cudaMallocHost()` that works similar to `cudaMalloc()` but allocates a page-locked memory on the host side. It's required for arrays like `h_xs` and `pready->pframe->data`. If it's pretty easy to be done for `h_xs` that is an array directly managed by the code, how is possible to do that for the pframe's data that is something managed by OpenCV? As said at the beginning, the *Mat* object offers a good option, the one to select an external pointer as data array.

```
uint8_t *h_frame;
cudaMallocHost((void **)&h_frame, total * sizeof *h_frame);
cudaMallocHost((void **)&pready->h_xs, total * sizeof *pready->h_xs);

// init of Mat object with page-locked h_frame ptr
pready->pframe = new Mat(ctx.sampleMat->rows,ctx.sampleMat->cols,
    ctx.sampleMat->type(), h_frame);

```

Without touching anything else, this optimization is able to achieve a good result. The video streaming is elaborated at around 22 fps with an average `for` time of 43.0 ms and an average `read` time of 0.0 ms, and the comparison with the previous case can be seen in Figure 6.

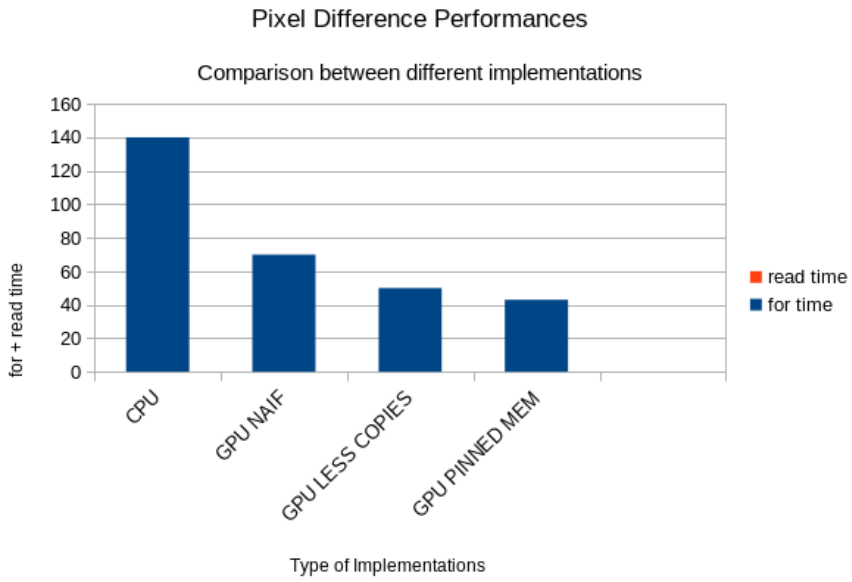


Figure 6: *Pixel Difference Comparison*

The results of the *nvprof* profiler are shown in Table 6 and is noticeable the big improvement in terms of performances for what concerns the HtoD transfer. In fact, it's about 1/4 the original one.

Operation	Avg (ms)	Min (ms)	Max (ms)
Kernel	43.386	38.093	49.607
DtoH	0.0102	0.0008	1.0466
HtoD	0.6740	0.6096	2.2818

Table 6: Profiling results

2.2.4 Kernel global memory access: the cherry on the cake

What remains now to further optimize is the kernel execution time. Doing a bunch of analysis, with a maximum number of threads of 1024 each kernel execute a *for loop* that lasts 6075 iterations. Each iteration access in memory three times in the most of the cases (supposing

$N_{pixelChanged} \ll N_{pixelNotChanged}$) and accesses in global memory are a real bit bottleneck for CUDA's kernel because usually the Global memory access time is $400 \sim 600$ cycles. Obviously, each byte needs to be esaminated so it's not possible to randomly cut accesses in memory. How this can be optimized?

The answer to this question is very simple but at the same time it's brilliant: by increase the size of each read or write in memory. By applying this concept, both the memory accesses and the loop iterations are reduced, so it's an optimization in both ways.

The first think to do is to rewrite the kernel in order to support this. The idea is to load into some GPU's registers some bytes regarding a portion of the current and previous frame under consideration by the for loop of a kernel. Then there is always need of a byte-wise substraction and comparison so a inner for loop will access byte-wise the word loaded into the register(s) and will do its computations.

The multi-byte memory access is done using the dynamic pointer cast feature, casting the vector pointers to the data type aimed, Therefore the new kernel implementation, using the most common data type that is the `int` one, is:

```
typedef int chunk_t;

__global__ void kernel2(
uint8_t *current, uint8_t *previous, uint8_t *diff,
int maxSect, unsigned int *pos, int *xs) {
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    unsigned int npos;
    int df;
    chunk_t cc, pc; // registers storing current copy, previous copy

    if (!x) *pos = 0;
    __syncthreads();

    int start = x * maxSect;
    int max = start + maxSect;

    #pragma unroll
    for (int i = start; i < max; i++) {

        cc = ((chunk_t *)current)[i];
        pc = ((chunk_t *)previous)[i];

        #pragma unroll
        for (int j = 0; j < sizeof cc; j++) {
            df = ((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j];

            if (df < -LR_THRESHOLDS || df > LR_THRESHOLDS) {
                npos = atomicInc(pos, 6220801);
                diff[npos] = df;
                xs[npos] = (i*sizeof cc) + j;
            } else {
                current[(i*sizeof cc) + j] -= df;
            }

        }

    }

}
```

The host code changes a little bit because maxAtTime now needs to be divided by the size of the `chunk_t` type. Because it's highly probable that the division results in a non integer value, it must be rounded to the nearest greater integer.

This means that by rounding up, each thread will overlap with the next thread of a few bytes. Actually it's not a problem because the probability that while the thread i writes at the very last bytes of its chunk the thread $i+1$ writes at the very beginning of its chunk is very low because

each thread is executed in parallel. The only thing that happens is that the computations on a certain amount of bytes is done two times but each time it's done on the same inputs and the result is always the same. In a low probability case where two writes at the same time happens, as discussed, the overhead in managing the concurrent write is lower than the one of adding instructions to check if the thread is writing outside its chunk (about 1 ms of difference in execution time).

```
int max4 = ceil(1.0 * maxAtTime / sizeof(chunk_t));

// CUDA APIs call
// ...
kernel2<<<1,nMaxThreads>>>(d_current,d_previous, d_diff, max4, d_pos, d_xs);
// ...
```

The only problem to manage by choosing the path of not adding any check inside the kernel is that the last kernel can write outside the memory. In order to avoid wrong memory accesses, the allocaiton of the memory on the GPU is done by considering this margin. In fact, a thread will write at most `sizeof(chunk_t)` bytes outside the memory, so each memory allocation is done by allocating an additional amount of bytes equal to `sizeof(chunk_t)` bytes:

```
uint8_t *h_frame;
cudaMallocHost((void **)&h_frame, total * sizeof*h_frame + sizeof(chunk_t));
cudaMallocHost((void **)&pready->h_xs,tot*sizeof*prdy->hxs+sizeof(chunk_t));

// init of Mat object with page-locked h_frame ptr
pready->pframe = new Mat(ctx.sampleMat->rows,ctx.sampleMat->cols,
    ctx.sampleMat->type(), h_frame);

```

The results of the *nvprof* profiler are shown in Table 7 and is noticeable the big improvement in terms of performances for what concerns the Kernel execution time. In fact, it's almost the 50% of the previous implementation. The average fps are 21, with a *for* time of 26.0 ms and a *read* time of 30 ms.

Operation	Avg (ms)	Min (ms)	Max (ms)
Kernel	23.700	21.590	24.671
DtoH	0.0079	0.0008	1.0009
HtoD	0.6483	0.6074	2.5641

Table 7: Profiling results

A further optimization can be done by looking at the new kernel implementation and noticing that only 2/3 of the memory accesses are optimized. The feedback write in fact is still done at each byte, and it can be easily optimized by modifying the `cc` register variable in the inner loop (byte wise) and save back the result in the global memory at the end of the inner loop only if needed (so if the `cc` variable has been modified).

The implementation is the following:

```
typedef int chunk_t;

__global__ void kernel2(
uint8_t *current, uint8_t *previous, uint8_t *diff,
int maxSect, unsigned int *pos, int *xs) {
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    unsigned int npos;
    int df;
    chunk_t cc, pc; // registers storing current copy, previous copy
    bool currUpdateRequired = false;

    if (!x) *pos = 0;
    __syncthreads();

    int start = x * maxSect;
    int max = start + maxSect;

```

```
    #pragma unroll
    for (int i = start; i < max; i++) {

        cc = ((chunk_t *)current)[i];
        pc = ((chunk_t *)previous)[i];

        #pragma unroll
        for (int j = 0; j < sizeof cc; j++) {
            df = ((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j];

            if (df < -LR_THRESHOLDS || df > LR_THRESHOLDS) {
                npos = atomicInc(pos, 6220801);
                diff[npos] = df;
                xs[npos] = (i*sizeof cc) + j;
            } else {
                ((uint8_t *)&cc)[j] -= df;
                currUpdateRequired = true;
            }

        }

        // storing in global memory only if needed
        if (currUpdateRequired) {
          ((chunk_t *)current)[i] = cc;
          currUpdateRequired = false;
        }
    }

}

```

The results of the *nvprof* profiler are shown in Table 8 and is noticeable another big improvement, reducing the kernel time by another 50%. The average fps are 24, with a *for* time of 11.5 ms and a *read* time of 40 ms. The overall comparison is shown in Figure 7.

Operation	Avg (ms)	Min (ms)	Max (ms)
Kernel	9.7944	8.7887	17.906
DtoH	0.0080	0.0008	1.2807
HtoD	0.6443	0.5969	2.9633

Table 8: Profiling results

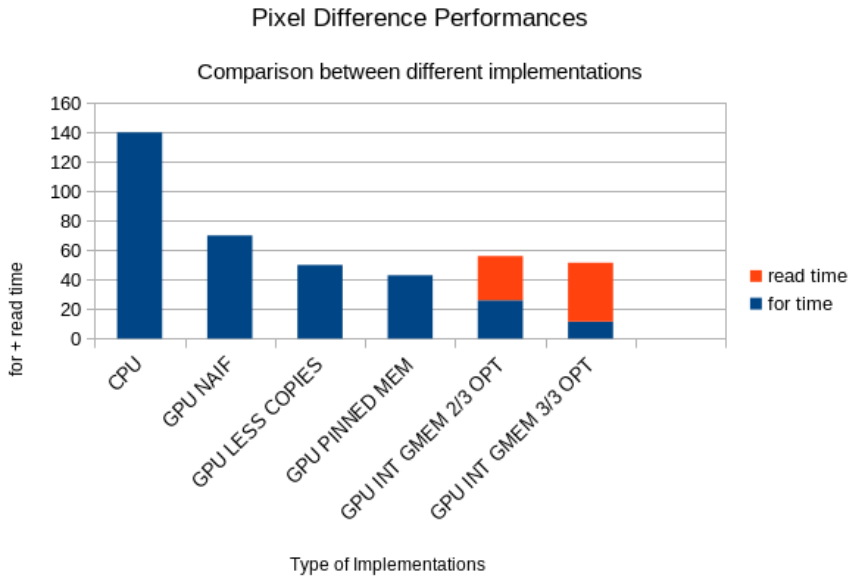


Figure 7: *Pixel Difference Comparison*

The overall situation can be optimized even more. CUDA offers the possibility to perform vectorized memory accesses by using *vectorized data types*. What are them and why they are so important?

First of all, these special data types are of different types depending on their reference type. For example there is `int2`, `float3`, `ulong2` and so on. Looking at the public header files they are defined as follow (taking `int4` as an example):

```
struct __device_builtin__ __builtin_align__(16) int4 {
    int x, y, z, w;
};

typedef __device_builtin__ struct int4 int4;
```

Thanks to the fact they they are `__device_builtin__`, they are identified by the CUDA compiler that will optimize the generated instructions by reducing them and using special *vectorized instructions* capable of a larger transaction in the memory. So thanks to these instructions is possible to load big chunks of data at a time with less instructions. Please note that a common struct with 16 int variables as members will not perform well as expected because int4 is a builtin type so recognized by the compiler and vectorized instructions will be used by activating some internal magic in the compiler itself.

The usage of these vectorized instructions is easily seen at the assembly level. The kernel seen before with a classical `int` data type is translated into these instructions:

```
/*00c8*/ MOV R3, R5;
/*00d0*/ LD.E R18, [R10];
/*00d8*/ LD.E R14, [R2];
/*00e0*/ SSY 0x240;
```

The two LD.E instructions are the one corresponding to the two reads from the global memory into the `cc` and `pc`. The LD.E is a scalar instruction that will load a word from memory. Using instead `int4` builtin type, the disassembly code will be:

```
/*00c8*/ MOV R3, R17;
/*00d0*/ MOV R12, R0;
/*00d8*/ LD.E.128 R4, [R2];
/*00e0*/ SSY 0x360;
/*00e8*/ LD.E.128 R8, [R12];
```

Here, the two LD.E.128 are vectorized instructions. By using the vectorized load and store instructions LD.E.{64,128} (and the counter part ST.E.{64,128}) operations like load and store data can be done in 64- or 128-bit widths. Using vectorized loads reduces the total number of instructions, reduces latency, and improves bandwidth utilization.

At this point, the *nvprof* profiler is run and the results are shown in Table 9 and is noticeable a really big improvement in the computation, reducing the kernel time by another 60%. The average fps are 26, with a *for* time of 4.7 ms and a *read* time of 37 ms.

Operation	Avg (ms)	Min (ms)	Max (ms)
Kernel	3.4197	2.9266	9.0223
DtoH	0.0061	0.0008	1.1601
HtoD	0.6387	0.5957	2.5595

Table 9: Profiling results

Even if the 128 bit width is the maximum width support, it's possible to optimize the resulting code even more at the instruction level by using a larger builtin types like `long4`. As can be intuitable, it will be translated into 2 instructions. This helps introducing a slightly optimization by reducing the memory access frequency, achieving an average reduction of 0.4 ms in the kernel execution time.

The resulting assembly code is the following one, where is possible to notice two loads at a time with the first one loading at offset `0x0` as the normal one and the second one loads at offset `0x10` that is 128 bit far from the first load, thus with two instructions a total of 256 bit or 32 byte are loaded, that is the size of a `long4` data type.

```
/*00c8*/ MOV R3, R13;
/*00d0*/ LD.E.128 R8, [R20];
/*00d8*/ LD.E.128 R4, [R20+0x10];
/*00e0*/ SSY 0x1e0;
/*00e8*/ LD.E.128 R12, [R2];
/*00f0*/ LD.E.128 R16, [R2+0x10];
```

The overall comparison is shown in Figure 8.

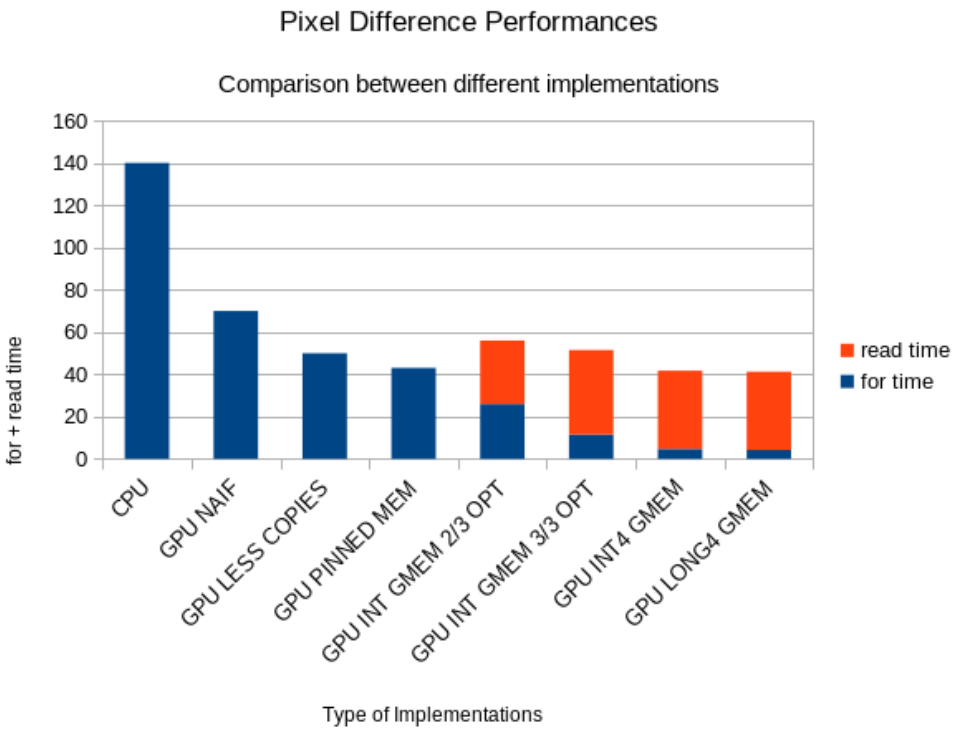


Figure 8: *Pixel Difference Comparison*

In other words, by using *vectorized instructions* the performance improvement comes from efficiency gains, in two ways:

1. At the instruction level, a multi-word vector load or store only requires a single instruction to be issued, so the bytes per instruction ratio is higher and total instruction latency for a particular memory transaction is lower.

2. At the memory controller level, a vector sized transaction request from a warp results in a larger net memory throughput per transaction, so the bytes per transaction ratio is higher. Fewer transaction requests reduces memory controller contention and can produce higher overall memory bandwidth utilization.

2.2.5 Using of shared memory for frequent accesses: is it avoidable?

Unfortunately, the algorithm and the relative CUDA kernel implementation doesn't show up any pattern of frequent access of the same data over the time, this is because the kernel works everytime on two frames and they differs continuously with respect to the previous kernel execution.

Looking at the last and very optimal kernel implementation, the only frequent global memory access where the same data is read and maybe updated is the one regarding the count of pixel

changed or `h_pos`. This means that it's reasonable to think to execute the increment operation `atomicInc()` in the shared memory and then tranfer the final result in the global memory in order to have only one access in global memory per thread with respect to the pixel count variable.

A possible implementation can be the following one:

```
typedef long4 chunk_t;

__global__ void kernel2(
uint8_t *current, uint8_t *previous, uint8_t *diff,
int maxSect, unsigned int *pos, int *xs) {
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    unsigned int npos;
    int df;
    chunk_t cc, pc; // registers storing current copy, previous copy
    __shared__ unsigned int s_pos[1];

    if (!x) *s_pos = 0;
    __syncthreads();

    int start = x * maxSect;
    int max = start + maxSect;

    #pragma unroll
    for (int i = start; i < max; i++) {

        cc = ((chunk_t *)current)[i];
        pc = ((chunk_t *)previous)[i];

        #pragma unroll
        for (int j = 0; j < sizeof cc; j++) {
            df = ((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j];

            if (df < -LR_THRESHOLDS || df > LR_THRESHOLDS) {
                npos = atomicInc(s_pos, 6220801);
                diff[npos] = df;
                xs[npos] = (i*sizeof cc) + j;
            } else {
                current[(i*sizeof cc) + j] -= df;
            }

        }

    }

    __syncthreads();
    if (!x) *pos = *s_pos;

}

```

Notice the use of the function `__syncthreads()` at the end. It is used to wait for all threads to finish in order to be sure that all the `atomicInc()` instructions have been executed. After this operation, the global memory variable `pos` is updated with the value stored in the shared memory variable `s_pos` by the first thread, the one with $id = 0$.

Unfortunately the result is something unexpected: the performance are exactly the same or a bit worse than the previous implementation. This is mainly for two reasons:

1. As explained in section 2.2.4, the part of code where the `atomicInc()` function is executed is the one with a lower execution frequency because it's supposed that between a frame and another only a bunch of pixels changes so it's not worth trying to improve it that requires adding other instructions (for example the one at the end of the kernel of synchronization) that only adds overhead.

2. Atomic instructions are very particular: they rely on the bottom hardware for executing memory operations in an atomic way.

About the last point, atomic memory operations are very important in parallel programming, allowing concurrent threads to correctly perform read-modify-write operations on shared data.

About the importance of how these kind of operations are implemented in CUDA architectures, as an example the Kepler architecture can be analyzed. It significantly increased the throughput of atomic operations to global memory compared to the earlier Fermi architecture; however, both Fermi and Kepler implemented shared memory atomics using an expensive lock/update/unlock pattern. So they were implemented totally in software.

However, Maxwell architecture (the GPU on the *Jetson Nano* used for this projct is based on this architecture) improved this by implementing native hardware support for shared memory atomic operations for 32-bit integers, and native shared memory 32-bit and 64-bit compare-and-swap (CAS), which can be used to implement other atomic functions with reduced overhead (compared to the Fermi and Kepler methods which were implemented in software).

So atomic operations have not been an easy thing. They are very performant for working in the global memory and *only recently* they have been optimized to work in the shared memory with near the same performance compared to the global memory.

3 Image Text Overlay

Another part of the project under discussion is the ability to add a text overlay over an image. In this particular implementation, it supports ASCII characters over a single line of the image. In order to be sent over internet, it's executed before applying the algorithm for sent the frame over a socket, according to Chapter 2.

The final result can be seen in Figure 9:

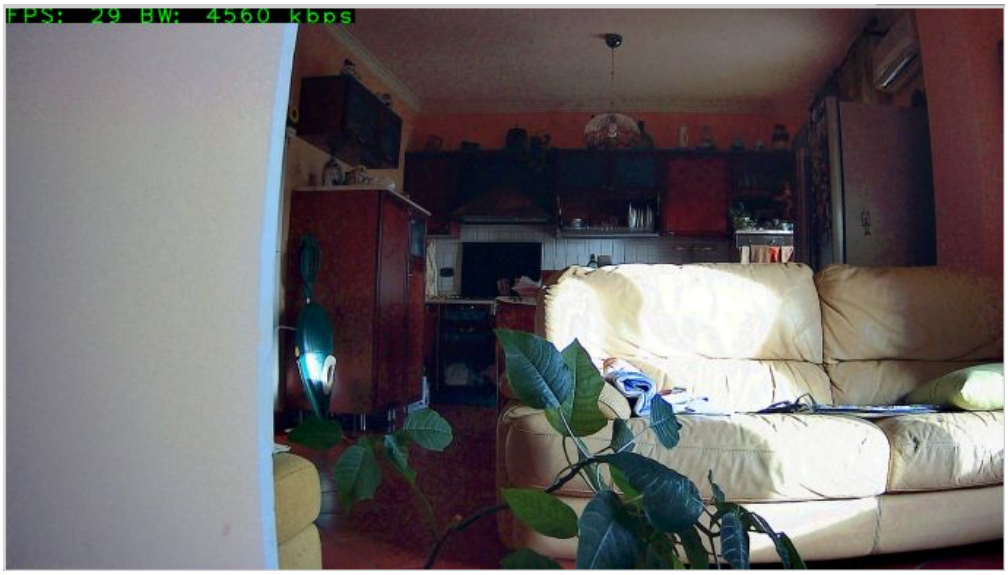


Figure 9: *Text Overlay*

The basic idea behind the implementation is the ability to generate a frame (made of RGB pixels) of an ASCII character, dynamically and based on an available font. Everything starts from the specification of the supported alphabet. In fact, not all ASCII characters are supported but only a limited set, specified during the compilation time, in order to have an efficient code in terms of memory occupied and execution time without wasting useful resources (both on the host side and on the device side) for characters that are never used.

After defining the alphabet, at the beginning of the program, it's possible to translate each character into a set of pixels composing it. This is done thanks to a series of OpenCV's function like `cv::getTextSize` that is able to rietrieve the frame dimensions (width and height) of a character and `cv::putText` that is able to put a text (or a character in this case) over a frame.

First thing that is done is to retrieve the dimension of a character. This is done only for a single character and it is supposed that all the other characters in the alphabet have the same size.

Than, an array storing all the frames one after another is created with a size that is equal to $n_{chars} \cdot Area_{char}$. A loop on the alphabet will fill this array with all the character' frames. The code to implement this is the following:

```
#define CHARS_STR "0123456789BFPSWbkps :"

auto txtsz = cv::getTextSize("A", cv::FONT_HERSHEY_PLAIN, 3, 2, 0);
int fullArea = 3 * txtsz.area();

uint8_t *charsPx = new uint8_t[(sizeof(CHARS_STR) - 1) * fullArea];
memset(charsPx, 0x0, (sizeof(CHARS_STR) - 1) * fullArea);
for (int i = 0; i < (sizeof(CHARS_STR) - 1); i++) {
  Mat pxBaseMat(txtsz.height, txtsz.width, base.type(), charsPx+i*fullArea);
  cv::putText(pxBaseMat, std::string(CHARS_STR).substr(i, 1),
      cv::Point(0, txtsz.height+1), cv::FONT_HERSHEY_PLAIN, 3,
      cv::Scalar(0, 255, 0), 2, cv::LINE_AA);
}
```

When the array is finally full filled, it's ready to be copied onto the GPU memory. This operation is done only at the beginning, so it's ininfluent from the point of view of the performances:

```
uint8_t *d_charsPx;
int totcpy = fullArea * sizeof *d_charsPx * (sizeof(CHARS_STR) - 1);
cudaMalloc((void **)&d_charsPx, totcpy * sizeof *d_charsPx);
cudaMemcpy(d_charsPx,charsPx,totcpy*sizeof*d_charsPx,cudaMemcpyHostToDev);
```

3.1 GPU Implementation

After the initialization phase, is now time to give a look at how each character frame is placed onto the image. It's a replacement of the pixels of the frame under analysis with the one of the corresponding character.

The first thing to do is to decide how many kernel to launch and how many pixel each kernel needs to manage. By deciding to assign to each kernel a certain number of pixel in a row-major fashion, and by considering the fact that the frame size of each character is fixed but can change if the font is changed, a portion of the code is used to decide in a dynamic way the two values: number of kernels and pixels for each kernel. It's decided by dividing the total number of bytes of a character frame by the number of frame and if it's a multiple of the size of `chunk_t` type that for a first analysis will be a `uint8_t`, then the number of threads found is the right one. The code is the following:

```
typedef uint8_t chunk_t;

int nThreadsToUse = nMaxThreads;
int eachThreadDoes = 1;
for (int i = nMaxThreads; i > 0; i--) {
  float frac = fullArea / (i * 1.0);
  int frac_int = fullArea / i;
  if ((int)ceil(frac) == frac_int && frac_int % sizeof(chunk_t) == 0) {
    nThreadsToUse = i;
    eachThreadDoes = frac_int / sizeof(chunk_t);
    break;
  }
}
```

3.1.1 Naif version

After the setup time, its the turn of the computation step. This is the core, where each character of the string to print is analyzed before applying the streaming algorithm. This is done on the host side and not on the device side in order to avoid the copy HtoD of the string.

The advantage is given by the usage of async operations, so while the frame is copied on the GPU, the host scans each character and a new kernel is fired in an async way in order to print each character one after the another.

```
// Copy in the current pointer and run
cudaMemcpyAsync(d_current,pready->pframe->data,total,cudaMemcpyHostToDev);

for (int offset = 0, j = 0; j < text.length(); j++, offset+=txtsz.width*3) {
  int idx;
  for (int i = 0; i < (sizeof(CHARS_STR) - 1); i++) {
    if (CHARS_STR[i] == text.at(j)) {
      idx = i;
      break;
    }
  }

  kernel_char<<<1, nThreadsToUse>>>(d_current, d_charsPx + idx * fullArea,
      eachThreadDoes, offset, 3 * txtsz.width, 3 * pready->pframe->cols);
}
```

The basic implementation of the kernel is the following:

```
__global__ void kernel_char(
uint8_t *current, uint8_t *matrix, int N, int offset,
int matrixWidth, int currWidth) {
    int thid = threadIdx.x + blockIdx.x * blockDim.x;

    int start = thid * N;
    int max = start + N;

    for (int i = start; i < max; i++) {
        int x = offset + i % matrixWidth;
        int y = i / matrixWidth;
        current[y * currWidth + x] = matrix[i];
    }

```

By running the profiler for a fixed time of 30 seconds, the kernel takes an average time of 2.981 us so it occupies the 1.53% of the total time.

A comparison chart is show in Figure 10:

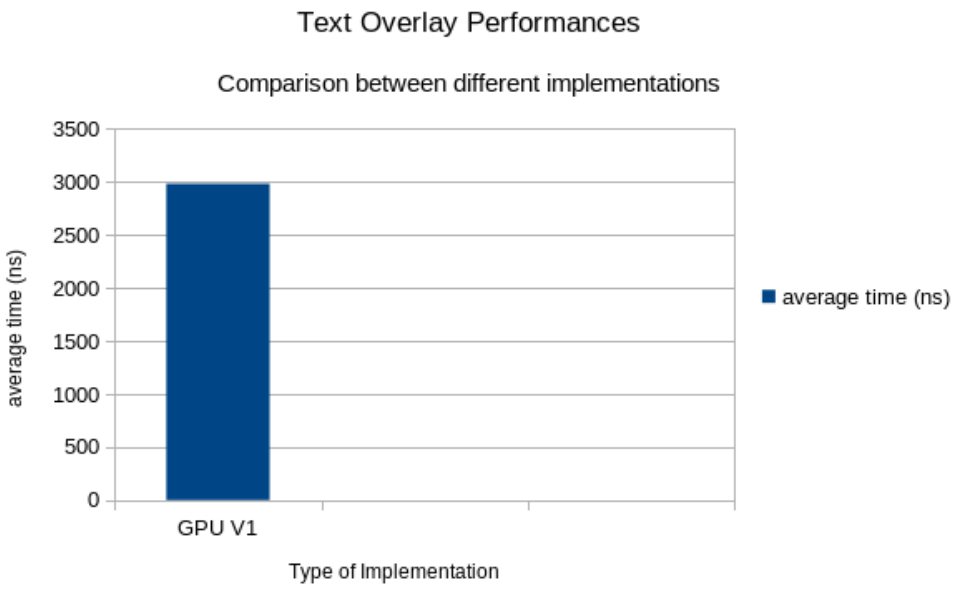


Figure 10: *Text Overlay Comparison*

3.1.2 Exploiting the global memory access optimization

It's possible then to optimize the kernel from the point of view of the global memory access parallelism as explained in Section 2.2.4. By copying and modifing the same kernel structure, the resulting kernel is:

```
typedef long4 chunk_t;

__global__ void kernel2_char(
uint8_t *current, uint8_t *matrix, int N, int offset,
int matrixWidth, int currWidth) {
    int thid = threadIdx.x + blockIdx.x * blockDim.x;
    chunk_t cc;

    int start = thid * N;
    int max = start + N;

    for (int i = start; i < max; i++) {
        int reali = i * sizeof(chunk_t);
        int x = offset + reali % matrixWidth;
        int y = reali / matrixWidth;

        int idx = (y * currWidth + x) / sizeof(chunk_t);
        cc = ((chunk_t *)current)[idx];

        #pragma unroll
        for (int j = 0; j < sizeof cc; j++) {
            ((uint8_t *)&cc)[j] = matrix[reali + j];
        }

        ((chunk_t *)current)[idx] = cc;
    }

```

By running the profiler for a fixed time of 30 seconds, the kernel takes an average time of 3.778 us so it occupies the 1.91% of the total time.

A comparison chart is show in Figure 11:

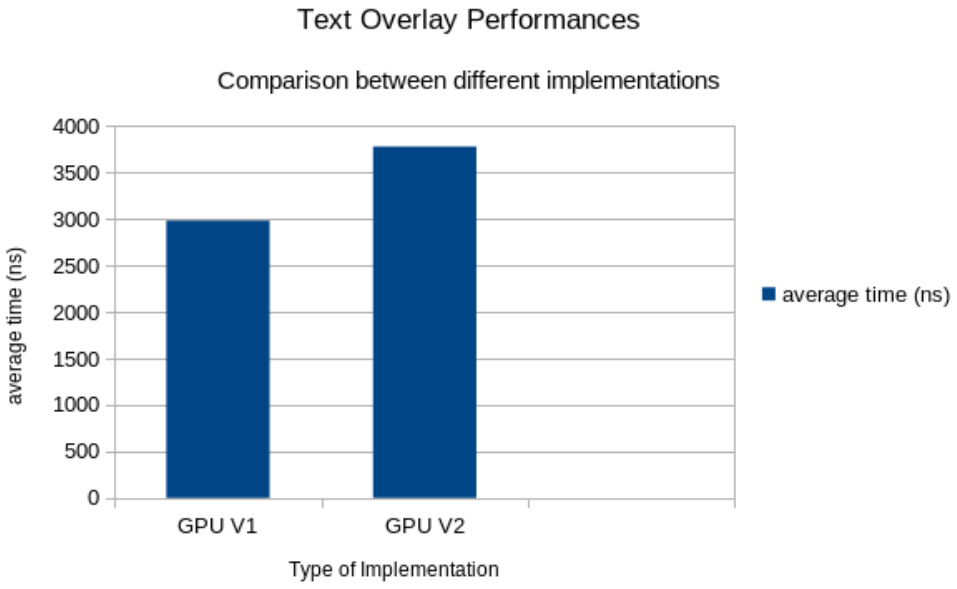


Figure 11: *Text Overlay Comparison*

As noticeable, the performance are worse. This is because the intermediate *for loop* can be compacted in a single instruction because it's only an assignment without other operations in the middle. The new kernel is then shorter and faster, as follow:

```
typedef long4 chunk_t;

__global__ void kernel2_char(
uint8_t *current, uint8_t *matrix, int N, int offset,
int matrixWidth, int currWidth) {
    int thid = threadIdx.x + blockIdx.x * blockDim.x;
    chunk_t cc;

    int start = thid * N;
    int max = start + N;

    for (int i = start; i < max; i++) {
        int reali = i * sizeof(chunk_t);
        int x = offset + reali % matrixWidth;
        int y = reali / matrixWidth;

        int idx = (y * currWidth + x) / sizeof(chunk_t);
        ((chunk_t *)current)[idx] = ((chunk_t *)matrix)[i];
    }

```

By running the profiler for a fixed time of 30 seconds, the kernel takes an average time of 1.868 us so it occupies the 1.01% of the total time. This means that the last version of the kernel is faster than the first one.

A comparison chart is show in Figure 12:

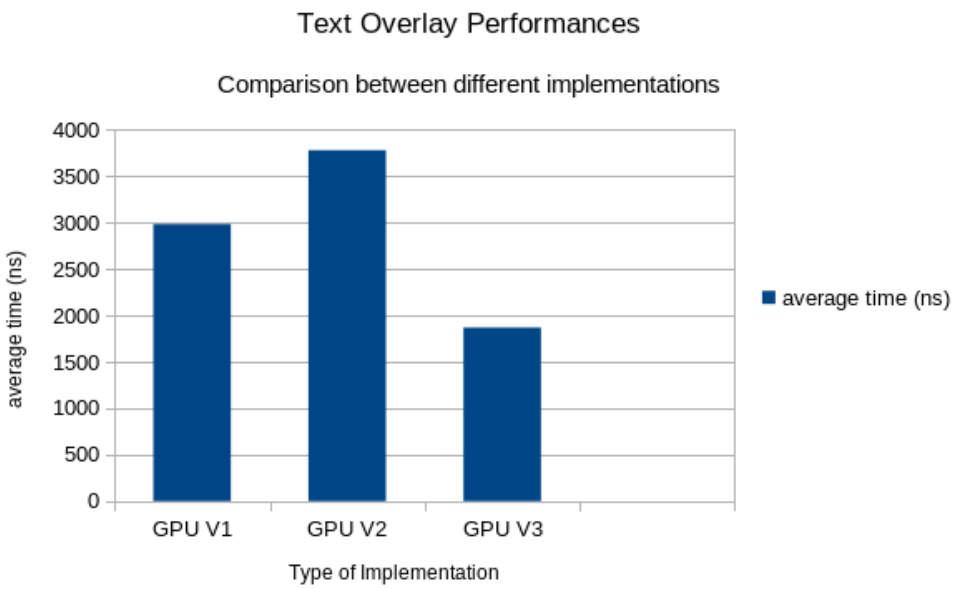


Figure 12: *Text Overlay Comparison*

3.1.3 Constant Memory instead of Global Memory: is it worth?

Because character frames are basically constant during the execution of the program, is it really necessary to use the global memory or another kind of memory can be exploited? If yes, is it really worth?

For this kind of purposes, CUDA devices offer the so called Constant Memory, that is a piece of memory on the GPU side that is constant. Because it's *constant* from the GPU point of view but it can modified by the Host, it seems to be perfect for this part of the project.

Usually it's large *64 KB* so supposing to have an alphabet whose frames doesn't exceed this memory limit, a constant variable (array) can be declared and initialized as follows:

```
__constant__ uint8_t c_matrix[(sizeof(CHARS_STR) - 1) * 3 * 32 * 28];

uint8_t *c_matrixPtr;
cudaMemcpyToSymbol(c_matrix, charsPx, totcpy);
cudaGetSymbolAddress((void **)&c_matrixPtr, c_matrix);
```

Leaving `kernel2_chars` as it is and using `c_matrixPtr` instead of d_charsPx in the kernel call and running the profiler, an unexpected value is returned by the profiler. The exeuction time is almost the same (faster only for a few nanoseconds) than the previous implementation and this is due to two reasons mainly:

1. the kernel is instruction-bound and not memory-bound
2. according to CUDA's programming guide: "*For all threads of a half-warp, reading from the constant cache is as fast as reading from a register as long as all threads read the same address. The cost scales linearly with the number of different addresses read by all threads.*". This means that the shared memory is really faster than the global memory but becuase the kernel is accessing every time a different address, the cost of access increases by reaching the cost of the global memory.

A comparison chart is show in Figure 13:

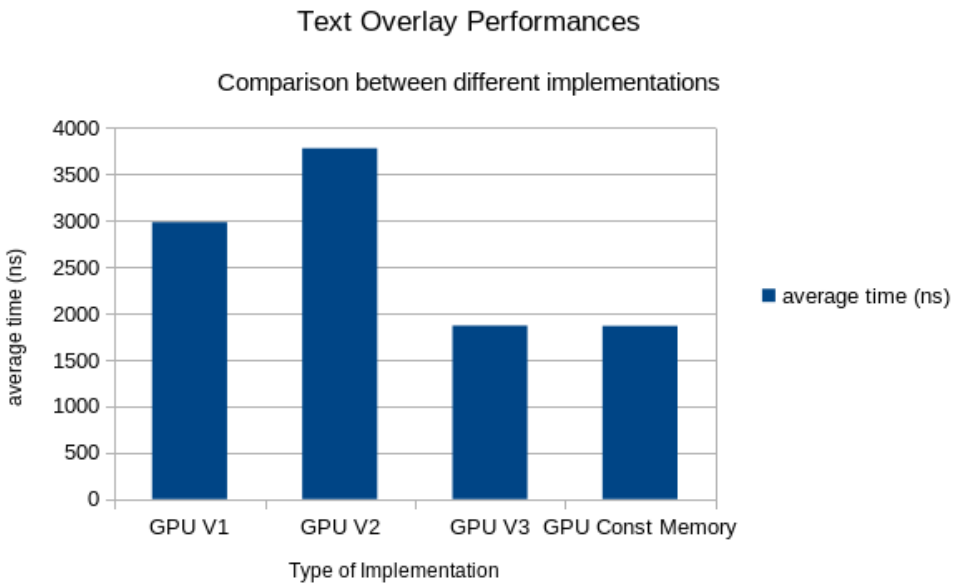


Figure 13: *Text Overlay Comparison*

4 HeatMap

Since the basis of this project is to send the pixel difference, the lower the dissimilarities the higher the bandwidth saving. In order to better visualize which pixels changes the most and the different magnitudes, an heatmap is generated with the current and the previous frames.

A heat map is nothing else than a data visualization technique that represents the magnitude of a measurement as colors in two dimensions, in this case an image. The idea is first compute the difference at the pixel level between two successive frames and then represent it using a color scale from blue to red, where blue means low difference and red high difference. The Image 14 shows a naif implementation of heat map, available at `heat_map_benchmark/v0.cu`, that generates in real time the image on the right.

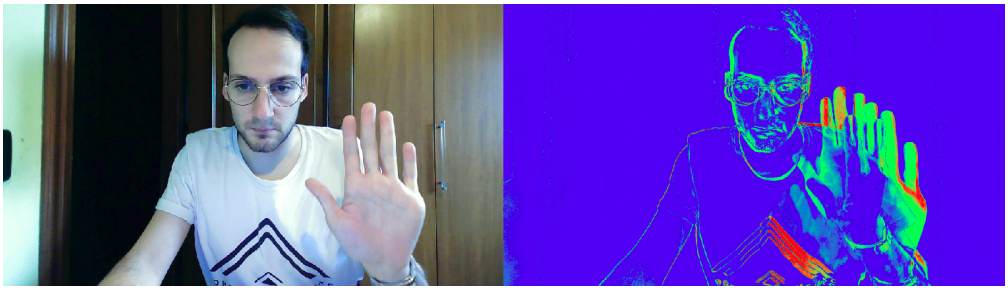


Figure 14: *Example of real-time heat map from webcam*

The basic steps for generating an heat map are the following:

- Take two frames from the webcam via OpenCV
- Compute the pixel difference
- Translate the difference into the corresponding color for the heat map
- Copy the result into a support buffer image and visualize it

All programs versions for the heat map are available in the `heat_map_benchmark` folder.

4.1 Heat map pixel mapping - CPU Naif implementation

The first *naif* version has been done on the CPU. This implementation can generate an heatmap in more or less 980ms, that is too much. This is also due to the complexity of the function itself that is used to map a normalized pixel difference to a blue-red scale and because the image is sequentially analyzed. In order to convert the difference of a pixel into the three color component (RGB), the usage of the *sine* function has been done. The difference of the each pixel has been computed as the sum of the absolute value of the difference of the single color component, as:

$$diff = abs(Previous[i,R] - Curr[i,R]) + abs(Previous[i,G] - Curr[i,G] + abs(Previous[i,B] - Curr[i,B])$$

Where $Previous[i,R]$ is the pixel red color component of the pixel at index i. In the worst case, a pixel can be turned from black to white or vice versa and so, the $0 \leq diff \leq 765$, that is $255 \cdot 3$. Then the $diff$ value is taken and normalize, by using 765 so that, $0 \leq diff_{norm} \leq 1$.:

$$diff_{norm} = \frac{diff}{765}$$

This value must be mapped into the tree RGB component of the heat map; the pixel must be more blue if the difference is more toward 0.0, yellow/green if near 0.5 and red if next to 1.0. The smoothed way to perform this task is to use three different sine functions, centered respectively on 0.0, 0.5 and 1.0 as in the following way:

$$\begin{aligned} \text{RED} \quad & sin(\pi \cdot diff_{norm} - \frac{\pi}{2.0}) \\ \text{GREEN} \quad & sin(\pi \cdot diff_{norm}) \\ \text{BLUE} \quad & sin(\pi \cdot diff_{norm} + \frac{\pi}{2.0}) \end{aligned}$$

The plot of the three sine functions is available at Figure 15.

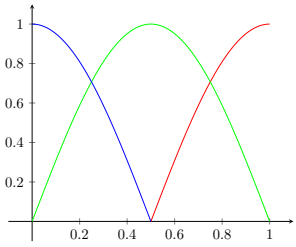


Figure 15: *Mapping function from pixel difference to RGB components*

The CPU implementation is the following:

```
struct HeatElemt {
  int r;
  int g;
  int b;
};

HeatElemt getHeatPixel(int diff){
  struct HeatElemt h;
  float diff1 = diff/(255.0*2.0);

  // Map the difference into the three color components
  h.r = min(max(sin(M_PI*diff1 - M_PI/2.0)*255.0, 0.0),255.0);
  h.g = min(max(sin(M_PI*diff1)*255.0, 0.0),255.0);
  h.b = min(max(sin(M_PI*diff1 + M_PI/2.0)*255.0, 0.0),255.0);
}

while (1) {
  cap >> image2;
  for (int y = 0; y < H; y++){
    for (int x = 0; x < W; x++){
      Vec3b & intensity = image1.at<Vec3b>(y, x);
      Vec3b a = image1.at<Vec3b>(y, x);
      Vec3b b = image2.at<Vec3b>(y, x);

      // Compute the absolute difference
      HeatElemt elem = getHeatPixel(abs(a.val[0] - b.val[0]) +
      abs(a.val[1] - b.val[1]) + abs(a.val[2] - b.val[2]));

      intensity.val[0] = elem.b;
      intensity.val[1] = elem.g;
      intensity.val[2] = elem.r;
    }
  }
  image1 = image2.clone();
}
```

In this CPU implementation, a loop is performed over each pixel of the two frames and the absolute difference computed. Another function, called `getHeatPixel` is used to convert that value to the heat map RGB color space. At this point, the original image is overwritten with the heatmap colors.

4.2 CUDA implementation

The idea is to rewrite what described in the previous section into a CUDA code. The GPU allows to run in parallel multiple instance of the same kernel, so that the execution can be done in parallel in order to speed up the computation.

From the perspective of the interaction, the GPU acts like an accelerator of the CPU, that ask it to execute the kernel by resorting to the following phases:

1. Copy the frames from the Host to the Device memory
2. Execute the kernel and wait its completion
3. Retrieve the result from the Device to the Host memory

This is exactly what the next section will explain. The kernel call allows to configure how many thread per kernel will be executed; obviously, depending on that number, the accessed locations must be defined accordingly. In fact, by defining K the number of thread lunched, each thread will work on a specific portion of the entire image:

$$\text{Thread portion dimension} = \frac{W \cdot H \cdot 3}{K}$$

Where W and H are the width and height of the image. The multiplication by 3 depends on the fact that each pixels is defined by three `uint8_t` datatype that each one of them defines a specific color.

Furthermore, from the memory allocation perspective of the GPU, the memory for the two frames and the heat map must be allocated. This is done only once at the startup of the program, thanks to the CUDA API.

```
// Pointer definition
uint8_t *d_current, *d_previous;
uint8_t *d_heat_pixels;

// Memory space reservion on GPU
cudaMalloc((void **)&d_current, W*H*C * sizeof *d_current);
cudaMalloc((void **)&d_previous, W*H*C * sizeof *d_previous);
cudaMalloc((void **)&d_heat_pixels, W*H*C * sizeof *d_heat_pixels);
```

Unfortunately, this type of the problem doesn't need any shared or constant memory because each location in the image (all three colors of all pixels) are accessed only once and using a shared memory would have only reduced the performance due to the overhead of the useless copy. Moreover, there are no data that are constant.

CUDA allows to get the information about the maximum number of thread by using the `cudaGetDeviceProperties` command and, by referring to the equation defined before, K can be set to that value. In fact, the kernel is launched with the following configuration:

- *Grid dimensions*: 1, 0, 0
- *Block dimensions*: K, 0, 0

Having the maximum number of threads in the `threads` variable, it's possible to call the kernel in the following way:

```
// Kernel per block computation
cudaGetDeviceProperties(&prop, 0);
int threads = prop.maxThreadsPerBlock;
int maxSection = (W*H*C)/threads;
```

An extensive analysis of the correct number of threads has been done at Section 4.3.

4.2.1 CUDA Naif implementation

The first implementation of the algorithm in CUDA is based on a 1:1 transposition of what done in the CPU, in the GPU. Always using OpenCV, two next frames are fetched, send to the kernel and the heat map computed.

```
// Copy from Host to Device
cudaMemcpy(d_prev, image1, W*H*C * sizeof *image1, cudaMemcpyHostToDevice);
cudaMemcpy(d_curr, image2, W*H*C * sizeof *image2, cudaMemcpyHostToDevice);

kernel<<<1, threads>>>(d_curr, d_prev, (W*H*C)/threads, d_out);

// Copy heat map from Device to Host
cudaMemcpy(heatmap, d_out, W*H*C * sizeof *heatmap, cudaMemcpyDeviceToHost);
```

This naif implementation implies two memory transfers (HostToDevice) for the previous and current frames and one DeviceToHost for the generated heat map.

In order to speed up the data management, instead of copying into a support array of `uint8_t` the entire two frames (previous and current), both frames are directly copied into the device buffers with the `cudaMemcpy` procedure.

Form the kernel perspective, there is a non-negligible complication with the respect to the CPU implementation. The CPU code is based on a single thread that iterates over the entire image in a sequential way, accessing one location after the other. For the GPU this is not the case, since now, each thread will work in parallel on a portion of the image that is long `maxSect`.

Due to this, each kernel must know exactly from which pixel start to retrieve the data and where to store the results. This is only a matter of index management; let's suppose to have frame with dimension 1920*1080*3. So, by using 1024 threads per block, each thread will work on:

$$maxSect = \frac{1920 * 1080 * 3}{1024} = 6075$$

This means that the first thread must work from 0 to 6074, the second one from 6075 to 12149 and so on. This can be simply achieved by giving an univocal index identifier to each kernel, that cam be generated as:

```
int x = threadIdx.x + blockDim.x * blockIdx.x;
```

The GPU implementation of the first CUDA kernel is the following:

```
__global__ void kernel(uint8_t *current, uint8_t *previous,
          int maxSect, uint8_t* d_heat_pixels) {

  // Index relative to the block
  int x = threadIdx.x + blockDim.x * blockIdx.x;

  // Start of the sector for this thread
  int start = x * maxSect;
  int max = start + maxSect;
  for (int i = start; i < max; i=i+C) {

    // Compute the pixel difference
    int pixelDiff = fabsf(current[i] - previous[i]) + fabsf(current[i+1]
        - previous[i+1]) + fabsf(current[i+2] - previous[i+2]);
    float diff1 = pixelDiff/(255*3.0);

    // Map different into the three color component
    int r = fminf(fmaxf(sinf(M_PI*diff1 - M_PI/2.0)*255.0, 0.0),255.0);
    int g = fminf(fmaxf(sinf(M_PI*diff1)*255.0, 0.0),255.0);
    int b = fminf(fmaxf(sinf(M_PI*diff1 + M_PI/2.0)*255.0, 0.0),255.0);
    d_heat_pixels[i] = b;
    d_heat_pixels[i+1] = g;
    d_heat_pixels[i+2] = r;
  }
}
```

After having run the `nvprof`, the main contributions to the execution time from the profiler are:

Type	**Time** (%)	**Avg**	**Name**
GPU activities	86.14	49.958	kernel
	9.38	2.5577ms	[CUDA memcpy HtoD]
	4.48	2.4427ms	[CUDA memcpy DtoH]
API calls	93.88	18.820ms	cudaMemcpy
	5.88	181.61ms	cudaMalloc

Table 10: *Profiling result v1.cu*

In fact, the cumulative time needed to copy all two frames into the device, execute the kernel and copy back the image in order to display it, takes approximately 57ms. From the GPU perspective, the average time to execute one single kernel is 49.958ms.

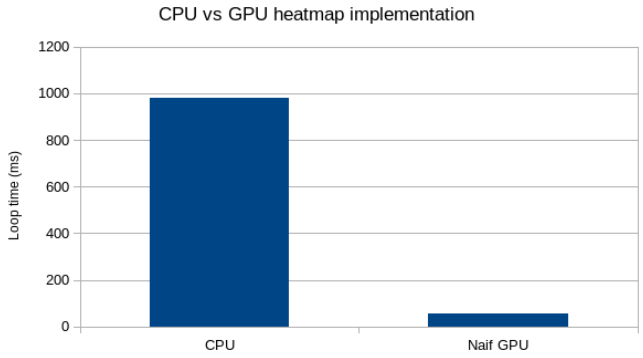


Figure 16: *Loop time CPU vs GPU*

4.2.2 CUDA switching frames

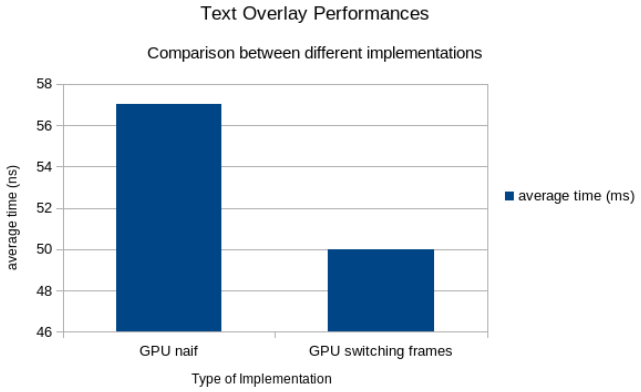
The first idea to reduce the time used for copying the frames from the Host to the Device. Instead of copying each time both the two frame, one of the two can be reused by only switching the two points, so that the one that previously was the current will became the previous; at this point, only the new frame must be copied into the device memory. This means that approximately the *CUDA memcpy HtoD* should be half of the previous time. This led to a `v2.cu` implementation, that exploit this pointer switching to reduce the memory transfer. The below code portion shows only the pointer switching in the loop used to fetch the frames.

```
while(1){
  // New frame fetch
  cap >> image2;

  // Pointer switching
  uint8_t* tmp = d_curr;
  d_curr = d_prev;
  d_prev = tmp;

  // Kernel call
  cudaMemcpy(d_curr, image2, W*H*C * sizeof *image2, cudaMemcpyHostToDevice);
  kernel<<<1, threads>>>(d_curr, d_prev, (W*H*C)/threads, d_out);
  cudaMemcpy(heatmap, d_out, W*H*C * sizeof *heatmap, cudaMemcpyDeviceToHost);

  image1 = image2.clone();
}
```

Type	Time (%)	Avg	Name
GPU activities	90.54	46.023	kernel
	4.85	2.4400ms	[CUDA memcpy HtoD]
	4.61	2.3453ms	[CUDA memcpy DtoH]
API calls	89.88	26.026ms	cudaMemcpy
	9.90	192.15ms	cudaMalloc

Table 11: *Profiling result v2.cu*

The average kernel execution time is in ms, and it is not a figure of interest, since it is more or less equal to the previous version. What is important is that now, the percentage of time used for the kernel is increase, due to the reduced *CUDA memcpy HtoD* (from 9.38% to 4.85%). This means that the GPU will analyze more frame in the same time frame. As we would expect, now the time needed to copy a frame from the memory to the device, execute the kernel and then retrieve the heat map takes about 50ms (12% faster).



4.2.3 CUDA Global memory access granularity

The problem with the generation of the heat map is that, the computation of the color must be done every time for all the pixels in order to compute the complete image.

Since each thread must perform 6075 iterations and need to write on the Global memory the same amount of times. In order to reduce the number of accesses to the Global memory, the idea was to access at the `int` level instead of the `byte` level. This means that frame information are still copied from host to device as arrays of *bytes* but they are accesses at the int level. So, if the current and the previous frames are passed as `uint8_t *current, uint8_t *previous`, the access is aligned at the 4 bytes. Another problem arises: the threads now access the memory with a granularity of 4 byte, but since the pixel difference needs needs only the first three bytes (RGB), in order to optimize and avoid to read twice from the memory, colors are updated only once every 3 bytes. So, when the position of the color in the image is the first, the colors of the heat map are set in the current and next two bytes of the output array, accordingly to the computed difference.

```
__global__ void kernel(uint8_t *current, uint8_t *previous,
          int maxSect, uint8_t* d_heat_pixels) {
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int start = x * maxSect;
  int max = start + maxSect;
  int cc, pc;
  for (int i = start; i < max; i++) {

    // Access one 4 byte at a time
    cc = ((int *)current)[i];
    pc = ((int *)previous)[i];
    int pixelDiff = 0;
    for (int j = 0; j < 4; j++) {

      // Conversion from difference to heat map only every 3 bytes
      if((i*4+j) % 3 == 0){
        int pixelDiff = fabsf(((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j]) +
        fabsf(((uint8_t *)&cc)[j+1] - ((uint8_t *)&pc)[j+1]) +
        fabsf(((uint8_t *)&cc)[j+2] - ((uint8_t *)&pc)[j+2]);
        float diff1 = pixelDiff/(255*2.0);
        int r = fminf(fmaxf(sinf(M_PI*diff1 - M_PI/2.0)*255.0, 0.0),255.0);
        int g = fminf(fmaxf(sinf(M_PI*diff1)*255.0, 0.0),255.0);
        int b = fminf(fmaxf(sinf(M_PI*diff1 + M_PI/2.0)*255.0, 0.0),255.0);
        d_heat_pixels[i*4+j] = b;
        d_heat_pixels[i*4+j+1] = g;
        d_heat_pixels[i*4+j+2] = r;

        // Reset the pixel difference
        pixelDiff = 0;
      }
    }
  }
}
```

Since now each threads works on 4 bytes at a iteration, the dimension of the data section that each block must work on is reduced by 1/4, as in the following way:

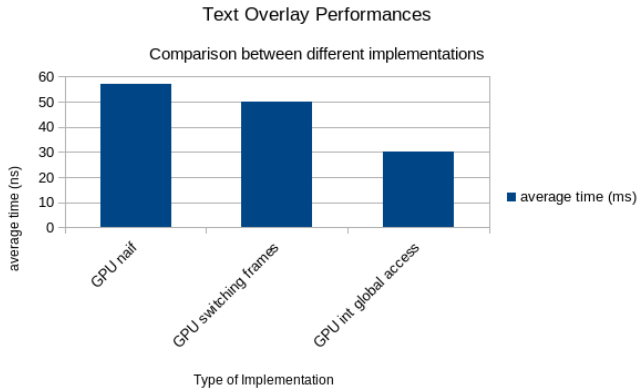
```
cudaGetDeviceProperties(&prop, 0);
int threads = prop.maxThreadsPerBlock;

cudaMemcpy(d_curr, image2, W*H*C * sizeof *image2, cudaMemcpyHostToDevice);

// Kernel call /4
kernel<<<1, threads>>>(d_curr, d_prev, ((W*H*C)/threads)/4, d_out);
cudaMemcpy(heatmap, d_out, W*H*C * sizeof *heatmap, cudaMemcpyDeviceToHost);
```

This led to another version, available at `v3.cu` allows to obtain the following results:

Type	**Time** (%)	**Avg**	**Name**
GPU activities	84.08	25.457ms	kernel
	8.14	2.4405ms	[CUDA memcpy HtoD]
	7.78	2.3549ms	[CUDA memcpy DtoH]
API calls	85.73	15.810ms	cudaMemcpy
	13.93	172.12ms	cudaMalloc

Table 12: *Profiling result v3.cu using int access level*

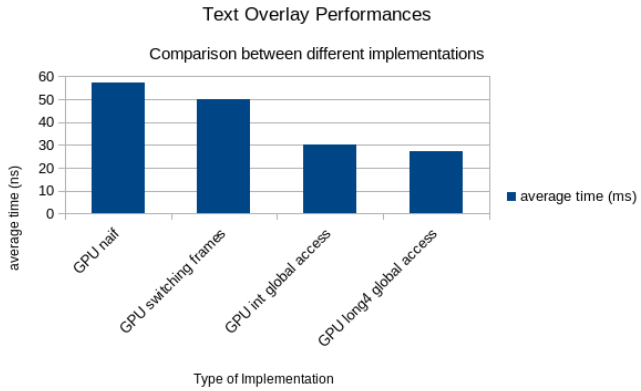
This version allows to copy the next frame from memory to device, execute the kernel and retrieve the result in about 30ms (about 40% of performance increase from `v2.cu`). This is highlighted in the table by the average time needed to execute the kernel itself, we went from 46.023ms to 25.457ms, thanks to the reduce access time to the memory.



This can be pushed even more, so that instead of accessing the images that are in the global memory, at the `int` level, they are accessed via a vectorized access. The complete explanation is available at Section 2.2.4. By the way, it is able to increase even more the performance, allowing to complete the for loop in around 27ms using `long4` data type.

Type	**Time** (%)	**Avg**	**Name**
GPU activities	82.83	23.589ms	kernel
	8.92	2.5160ms	[CUDA memcpy HtoD]
	8.24	2.3478ms	[CUDA memcpy DtoH]
API calls	85.42	14.903ms	cudaMemcpy
	14.18	165.80ms	cudaMalloc

Table 13: *Profiling result v3.cu using long4 access level*



4.3 Evaluation of the number of threads

For a first implementation, the number of thread has been set to the maximum allowable from the architecture, that is given by the `cudaGetDeviceProperties` CUDA function in order to make the program independent form the device used. For example, for the Jetson Nano, the maximum number of threads are 1024.

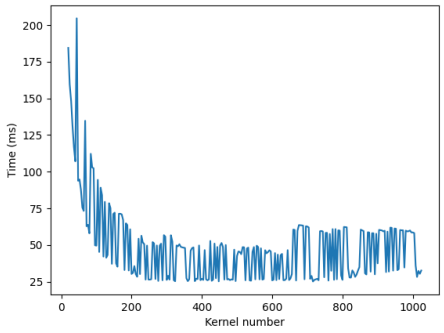
In order to understand how the number of threads impacts on the heat map generation, a bash script has been build in order to dynamically change the K parameter via compiler directive.

The number of threads must be a multiple of 4, so that the array of the pixels can be divided into portion in such a way that a pixel is not split between two kernel.

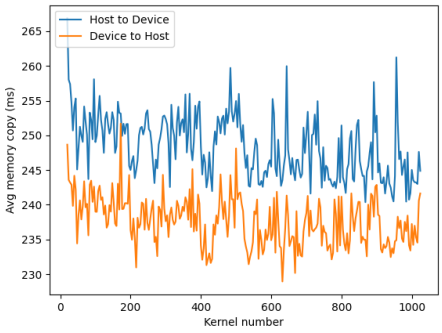
This is the bash code used to extract the *nvprof* information:

```
#!/bin/bash
for i in {1..256}
do
  k=$(( 4*i ))

  # Profiler call
  avg=‘sudo /usr/local/cuda/bin/nvprof ./heatMap $k 2>&1‘

  # Data extraction
  kern=‘echo "$avg" | grep kernel | awk ’{print $6}’‘
  kernt=‘echo "$avg" | grep kernel | awk ’{print $4}’‘
  hd1=‘echo "$avg" | grep "CUDA memcpy HtoD" | awk ’{print $4}’‘
  hd2=‘echo "$avg" | grep "CUDA memcpy HtoD" | awk ’{print $2}’‘
  dh1=‘echo "$avg" | grep "CUDA memcpy DtoH" | awk ’{print $4}’‘
  dh2=‘echo "$avg" | grep "CUDA memcpy DtoH" | awk ’{print $2}’‘
  echo "$k $all $kern $kernt $hd1 $hd2 $dh1 $dh2" >> times.txt
done
```

In this way, the output of the profiler and the time needed to copy the frame, generate the heat map and retrieve the result is parsed and wrote into a file called `times.txt`. Thanks to another script, the most useful data are plot, as below:



(a) Time needed to perform an heatmap depending on the number of kernel set



(b) Time needed to copy from Host to Device (blue) and from Device to Host (orange)

This is not the behaviour that we would have expected, beside the time needed to copy is more or less constant, by increasing the number of threads for that kernel we would expect that the time needed for the heat map computation would be lower. This is probably due to the fact that the *warp* has a fixed size of 32 threads and, even if by increasing the number of threads its execution time is lower, their management probably introduce too much overhead to obtain benefits.

Beside this, the time needed to perform an heatmap vs the number of threads, shows a peculiar behaviour. After about $N > 280$ the times tend to oscillate between more or less 50ms and 27ms. Even if this seems not a big difference, in the filed of real-time image processing, it's a huge improvement. In order to avoid errors, the same script has been run multiple times and the results is always the same. Since the internal infrastructure is a black box, the hardware probably manage in different ways the threads with the respect to their number.
The best observed thread configuration for the Jetson Nano, seems to be 716.

In fact, by running the same exact algorithm describe before but the number of threads is set to 718 (that is the best kernel accordingly to the plot above), the time needed to copy a frame, compute the heat map and then copy back the heat map matrix is about 27ms. This leads to a increasing of performance of 10%, that in this domain is not negligible.

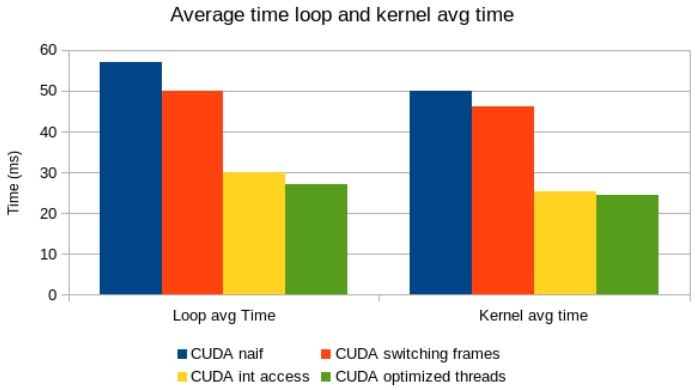
The following table shows some meaningful data extracted from the profiler:

Type	Time (%)	Avg	Name
GPU activities	84.30	24.380ms	kernel
	8.04	2.4599ms	[CUDA memcpy HtoD]
	7.66	2.383ms	[CUDA memcpy DtoH]
API calls	90.06	16.192ms	cudaMemcpy
	9.57	115.27ms	cudaMalloc

Table 14: *Profiling result v3.cu*

4.3.1 Heatmap Conclusions

By considering only the CUDA algorithm, the optimizations shows a decreasing fashion of the time needed to process the two frames and generate the heat map. The Loop average time, is the time needed to copy the image from the Host to the Device, summed to the heat map time computation and the time needed to copy it back. On the other hand, the plot on the right shows only the average kernel time. All data are in ms.



5 Noise visualizer

As already explained in the first sections, the algorithm is based on sending only the difference of all pixels whose color components are above a certain threshold.

In order to better visualize the noise in each frame taken from the webcam, all colors of all pixels that are above a certain threshold, are colored in red. The threshold is the same used by the kernel that computes all pixel differences. Obviously, the complexity of this computation is much easier than the previous one and the CPU allows to compute that kind of map in around 250ms.

Figure 18: *All non normalized pixel difference are turned to red if above a certain threshold, in this case 20. Beside the figure in the center, all red pixels can be identified as noise.*

The purpose of this new color mapping is completely different from the heatmap, since the latter one is used to understand the magnitude of the pixel difference while the second one (the black-red) is used to better visualize the noise pixels, by using a threshold.

Why is this necessary? This allows to have a visual evaluation of the noise filter, that is used to reduce smooth the noise in order to reduce the difference between two frames and reduce even

more the bandwidth. The noise filter will be explained in the next section.

First of all, we need to define what is the noise: by means of noise we are referring to all the random variations on the color information in an image, that is visible as grain in film and pixel level. The level of the noise mainly depends:

- Length of the exposure
- Sensitivity of the optical camera
- Brightness of the environment
- Temperature

In most of the cases, the webcam used at home is a cheap one that has an extremely small pixels in the camera sensor. There are mainly three types of noises: fixed pattern noise, random noise and banding noise. The common noise in our webcam is the one shown below. As you can see, the dark and less brightness parts of the image are characterized by some particles that shows different color from the whole context.

Figure 19: *Example of noise*

From our perspective this is a very bad behaviour due to the fact that the noise is random, and can change the value of the pixel at each frame. This leads to an high amount of pixel changed, even if the image is always the same. For this reason, *noise image filtering* is performed over the frame, before being analyzed by the CUDA kernel that computes the pixel differences.

The internal logic of a webcam allows to trim some parameters of the video capture directly by OpenCV. This is not a software level manipulation of the frame, but the internal control unit of the webcam, while sampling the frame, performs different behaviour depending on the selected configuration.

For this reason, the original image has been enhanced with a control panel, so that the user can change the brightness, the contrast, the saturation and the gain of the image. The Figure 20, shows the panel that is shown to the user. Even if it can be used to reduce the noise, by for example reducing the contrast, it allows the user to increase the quality of the send image.

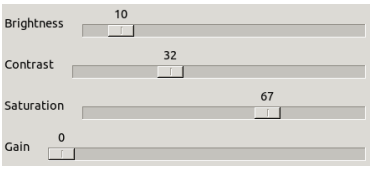


Figure 20: *Image Control panel*

5.0.1 Naif CPU implementation

The *naif* CPU implementation starts from the same program base used for the heat map. The entire image, pixel by pixel, is analyzed and if one of the color component is above the same threshold set for the pixel difference, that pixel is colored in red otherwise in black.

The code below is the C implementation of what explained before, where `LR_THRESHOLDS` is set to 20.

```
for (int y = 0; y < H; y++){
  for (int x = 0; x < W; x++){
    Vec3b & intensity = image1.at<Vec3b>(y, x);
    Vec3b a = image1.at<Vec3b>(y, x);
    Vec3b b = image2.at<Vec3b>(y, x);

    if(abs(a.val[0] - b.val[0]) > LR_THRESHOLDS ||
          abs(a.val[1] - b.val[1]) > LR_THRESHOLDS ||
          abs(a.val[2] - b.val[2]) > LR_THRESHOLDS)
    {
      intensity.val[0] = 0;
      intensity.val[1] = 0;
      intensity.val[2] = 255;
    } else {
      intensity.val[0] = 0;
      intensity.val[1] = 0;
      intensity.val[2] = 0;
    }
  }
}
```

The resulting image is the one at Figure 18. From the time perspective, as said before, being the operations for the noise visualizer less complex, the CPU is able to generate the image in around 250ms.

5.0.2 CUDA implementation

The same considerations and optimization steps done for the heat map can be done also for the generation of the frames that allows the user to better visualize the noise in the frames. So, starting from an already optimized memory management, the resulting kernel call is based as always on first copy the frame into the Device Global memory, call the kernel and then retrieve the image, as in the code below:

```
  cudaGetDeviceProperties(&prop, 0);
  int threads = prop.maxThreadsPerBlock;

  cudaMemcpy(d_curr, image2, W*H*C * sizeof *image2, cudaMemcpyHostToDevice);

  // Kernel call /4
  kernel_red<<<1, threads>>>(d_curr, d_prev, ((W*H*C)/threads/
                                (sizeof(chunk_t)), d_out);
  cudaMemcpy(heatmap, d_out, W*H*C * sizeof *heatmap, cudaMemcpyDeviceToHost);
```

In this case, the optimization described in the pixel difference generation, regarding the *vectorized access* to memory is also applied here, where a new type called `chunk_t` is defined. In this way more byte can be read with a single memory access.

```
typedef int4 chunk_t;

__global__ void kernel_red(uint8_t *current, uint8_t *previous,
          int maxSect, uint8_t* d_heat_pixels) {
      int x = threadIdx.x + blockDim.x * blockIdx.x;
  int start = x * maxSect;
  int max = start + maxSect;
  chunk_t cc, pc;
  uint8_t redColor = 0;
  int df;
  int size = sizeof(chunk_t);

  for (int i = start; i < max; i++) {
    cc = ((chunk_t *)current)[i];
    pc = ((chunk_t *)previous)[i];

    for (int j = 0; j < size; j++) {
      df = ((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j];

      if ((df < -LR_THRESHOLDS || df > LR_THRESHOLDS)){
        redColor = 255;
      }

```

```
24          if(((i*size)+ j ) % 3 == 2){
25          noise_visualization[(i*size)+j] = redColor;
26          redColor = 0;
27        } else {
28          noise_visualization[(i*size)+j] = 0;
29        }
30      }
31  }
```

By iteration on all the color components of all the pixels in the frame, the difference with the previous frame is checked. If it is above the threshold, the value of the red color for that pixel is set to 255.

Being the red component the third in a pixel, it is set accordingly to the `redColor` variable only if the `i*size + j % 3 == 2` (the red color).

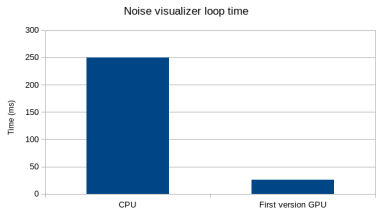
This implementation allows to copy the frame from Host to Device, compute the noise visualization and retrieve back the image from Device to Host memory in around 26ms.

The following table shows some meaningful data extracted from the profiler:

Type	Time (%)	Avg	Min	Max	Name
GPU activities	81.38	21.759ms	19.489ms	32.942ms	kernel_red
	9.59	2.5469ms	2.2134ms	6.8552ms	[CUDA memcpy HtoD]
	9.03	2.4141ms	2.1828ms	3.5408ms	[CUDA memcpy DtoH]
API calls	87.41	13.992ms	2.7579ms	156.01ms	cudaMemcpy
	12.13	194.86ms	3.8501ms	576.73ms	cudaMalloc

Table 15: *Profiling result v1.cu for noise visualizer*

As you can see, the average time needed to compute the kernel is less with the respect to the heat map. This is due to the simplicity of the function that only need to check the difference against a threshold. As you can see, the GPU acceleration has a huge impact over the computation.



5.0.3 CUDA reduction of global memory accesses

The fact that the only pixels that changes is the red one, the previous CUDA kernel can be optimized even more in order to reduce the number of access to the global memory. In fact, if we set assume that at the beginning all pixels in the `d_heat_map` array are all completely black, the algorithm can reduce the memory access of on third since it will work only on a the red color component. As before, if the difference is above a threshold, the red pixel is turned to red, or to black otherwise.

The resulting kernel is the following:

```
__global__ void kernel_red(uint8_t *current, uint8_t *previous,
  int maxSect, uint8_t* d_heat_pixels) {
    int x = threadIdx.x + blockDim.x * blockIdx.x;
    int start = x * maxSect;
    int max = start + maxSect;
    chunk_t cc, pc;
    uint8_t redColor = 0;
    int df;
    int size = sizeof(chunk_t);

    for (int i = start; i < max; i++) {
      cc = ((chunk_t *)current)[i];
      pc = ((chunk_t *)previous)[i];
```

```
14
15          for (int j = 0; j < size; j++) {
16            df = ((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j];
17
18            if ((df < -LR_THRESHOLDS || df > LR_THRESHOLDS)){
19              redColor = 255;
20            }
21
22            if(((i*size)+ j ) % 3 == 2){
23            noise_visualization[(i*size)+j] = redColor;
24            redColor = 0;
25          }
26        }
27  }
```

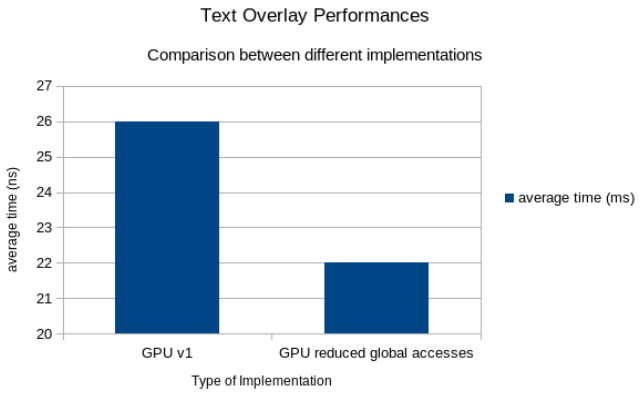
As you can notice, the only difference is that the else condition is the last if statement has been removed. This means that the green and blue component are not modified, thus the global memory is left as is for these two bytes.

This allowed to increase the performance of the kernel even more; as you can see from the table below the kernel now lasts in average 16.684ms, that is a 23% faster. The entire loop is done in about 22ms.

Moreover, the test must be performed in the same conditions because, depending on the video captured, it can show different number of changed pixels with the value of the difference above or below the threshold. For this reason, the *nvprof* has been run while the webcam was covered with a black tape. Due to the fashion of the algorithm, also the minimum and maximum values have been included in the table below.

Type	**Time** (%)	**Avg**	**Min**	**Max**	**Name**
GPU activities	75.63	16.684ms	16.372ms	35.942ms	kernel_red
	12.37	2.6294ms	2.2265ms	6.3841ms	[CUDA memcpy HtoD]
	12.00	2.6294ms	2.2265ms	6.3841ms	[CUDA memcpy DtoH]
API calls	86.00	11.610ms	2.6999ms	41.414ms	cudaMemcpy
	13.56	183.71ms	3.7352ms	543.50ms	cudaMalloc

Table 16: *Profiling result v2.cu for noise visualizer*



The code of this and the previous implementation has been disassembled using the NVIDIA tools *cuobjdump* and *nvdisasm*; the last one is also needed to get information about the control flow. In order to avoid that the *nvcc* compiler to avoid, with very limited exceptions, dead-code eliminations and register-spilling optimizations. For this reason the code has been compiled with the `-O0` flag.

The generation of the assembly is:

```
1 cuobjdump ./a.out -xelf all
2 nvdisasm v1.sm_30.cubin
```

By looking carefully at the assembly code of the first implementation, it's possible to understand the correlation between the assembly code and the instructions. As we would have expected, in the code of the first implementation are present two `ST.E.U8` instructions, used to perform the if and else case.

The following is an extract of the assembly that manage the if statement on the first CUDA implementation.

```
   // Perform comparison
   /*0ab0*/ ISETP.EQ.X.AND P0, PT, R5, RZ, PT;
   /*0ab8*/ PSETP.AND.AND P0, PT, !P0, PT, PT;
   /*0ac8*/ SSY `(.L_16);

   // Go to if
   /*0ad0*/ @P0 BRA `(.L_17);

   // Go to else
   /*0ad8*/ BRA `(.L_18);


// IF
.L_17
   ...
   /*0e10*/ ST.E.U8 [R4], R0;
   /*0e28*/ NOP.S (*"TARGET= .L_16 "*);

// ELSE
.L_18:
   ...
   /*0c50*/ ST.E.U8 [R4], R0;
   /*0c88*/ NOP.S (*"TARGET= .L_16 "*);


// Management of the loop control flow
.L_16
   ...
```

Figure 21: Assembly extract of the first CUDA implementation of the noise visualizer

So, label `.L_17` and `.L_18` are used to identify the branch that is used to write 0 or 255, accordingly to the fact that it is or not the red component and only if above the threshold.

On the other hand, the assembly of the second implementation shows only one instance of the store byte instruction, as we would have expected:

```
  // Perform comparison
  /*0aa8*/ IADD32I RZ.CC, R4, -0x2;
  /*0ab0*/ ISETP.EQ.X.AND P0, PT, R5, RZ, PT;
  /*0ab8*/ PSETP.AND.AND P0, PT, !P0, PT, PT;
  /*0ac8*/ PRMT R0, R29, 0x7610, R0;
  /*0ad0*/ PSETP.AND.AND P0, PT, P0, PT, PT;
  /*0ad8*/ PRMT R0, R0, 0x7610, R0;
  /*0ae0*/ SSY `(.L_16);
  /*0ae8*/ @P0 NOP.S (*"TARGET= .L_16 "*);
  /*0af0*/ BRA `(.L_17);


// IF
.L_17
  ...
  /*0e10*/ ST.E.U8 [R4], R0;
  /*0e28*/ NOP.S (*"TARGET= .L_16 "*);


// Management of the loop control flow
.L_16
  ...
```

Figure 22: Assembly extract of the second CUDA implementation of the noise visualizer

5.0.4 Overlapping implementation

The fact that the we can have a frame that changes completely with the respect to the previous one, implies that not only the black pixels must be turned in red if above a certain threshold, but also the red ones that are below must be changed to black. This means that we if we have an image $1920 * 1080 * 3$ we can't go below $1920 * 1080$ access to Global Memory.

A second approach resort to not use the output `d_heat_pixels` as output array but instead write directly over the *previous* frame. In this way, the red pixel will be placed over the frame itself; this is a completely different representation but, if we can modify the original frame (that will be discharged at the end of the loop), this solution is feasible. The result is visible at Figure 23.

Figure 23: *Red pixels over the frame*

This has be done to overcome the need of setting a pixel to black when the difference between the pixels is below the threshold. With this approach, the number of write to the Global Memory should corresponds to the number of pixels whose difference is above the threshold. Hypothetically, two identical frames should not need any write access to the Global Memory.

The kernel has been modified in the following way, so that the red color is set only if at least one color component of that pixel is above the threshold:

```
__global__ void kernel_red(uint8_t *current, uint8_t *previous,
int maxSect) {
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int start = x * maxSect;
  int max = start + maxSect;
  chunk_t cc, pc;

  bool toUpdate = false;
  for (int i = start; i < max; i++) {
```

```
10
11      cc = ((chunk_t *)current)[i];
12      pc = ((chunk_t *)previous)[i];
13      for (int j = 0; j < sizeof cc; j++) {
14        int8_t df = ((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j];
15
16        if (df < -LR_THRESHOLDS || df > LR_THRESHOLDS) {
17          toUpdate = true;
18        }
19
20        // Update only if needed
21        if (toUpdate && (i*(sizeof cc)+j) % 3 == 2){
22        previous[i*(sizeof cc)+j] = 255;
23        toUpdate = false;
24      }
25    }
26  }
```

Obviously, also the kernel call must be modified, so that instead of copying back data from the `d_heat_map` the data are retrieved from the previous frame, that has been overwrite during the kernel execution:

```
cudaMemcpy(d_current, image2.data, W*H*C * sizeof *image2.data,
                          cudaMemcpyHostToDevice);
kernel<<<1, threads>>>(d_current, d_previous, (((W*H*C)/threads)/
                          (sizeof(chunk_t))));

// Rewrite d_previous frame
cudaMemcpy(res.data, d_previous, W*H*C * sizeof *res.data,
                         cudaMemcpyDeviceToHost);
```

This implementation, in the same test condition that has been used also for the other two implementations of the noise visualizer, allows to perform the copy of the frame from Host to Device, compute the noise visualization and copy it from the Global memory of the Device to the Host in around 16.5ms.

These are the results from *nvprof*:

Type	**Time** (%)	**Avg**	**Min**	**Max**	**Name**
GPU activities	53.05	7.5647ms	4.9421ms	51.426ms	kernel_red
	23.69	3.3450ms	2.4960ms	6.6161ms	[CUDA memcpy HtoD]
	23.26	3.3160ms	2.8359ms	4.2297ms	[CUDA memcpy DtoH]
API calls	81.86	7.7539ms	2.9697ms	57.082	cudaMemcpy
	17.51	166.67ms	5.4298ms	327.91ms	cudaMalloc

Table 17: *Profiling result v3.cu for noise visualizer, overwriting the red pixels over the previous frame*

As you can see from the time for the kernel execution, there is a huge improvement from the black and red implementation of about 25% more.

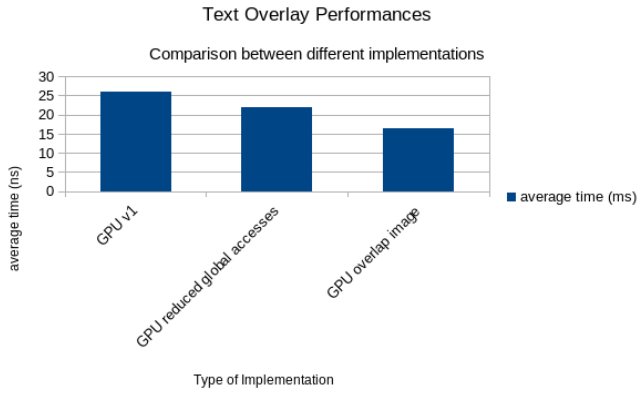


Figure 24: *Comparison between GPU versions for noise visualizer*

5.1 Fastest implementation

By exploiting the same idea described in the previous paragraph, the algorithm can be even more optimized from the point of view of the execution time. In fact, if instead of taking the two images and compute the difference at pixel level in order to decide if the pixel will be turned red or left as is, the difference computed from the dedicate kernel can be exploited.

By referring to the Section 2, the output of the kernel that computes the difference pixel by pixel in order to send the difference is stored into the `h_diff`. Moreover, an array called `h_xs` contains the locations in which the pixels are changed.

The idea is to exploit the `h_xs` array in order to turn red the corresponding pixels. Its length is defined by an integer value called `h_pos`. Having `nMaxThreads` threads and `h_pos` pixels that must be turned to red, the total threads can work on a portion of `h_xs` that is large:

$$maxSect = \frac{nMaxThreads}{h\_xs}$$

This implementation allows to have a huge speedup, since the `h_xs` has its corresponding vector already in the Device, called `d_xs`; for this reason, it's enough to simply pass the point, without previously perform any memory copy from Device to Host.
The final CUDA kernel call is like the following C code:

```
// Compute pixel difference
kernel2<<<1, nMaxThreads, 0>>>(d_current, d_previous, d_diff, max4, d_pos, d_xs);
cudaMemcpyAsync(&pready->h_pos, d_pos, sizeof *d_pos, cudaMemcpyDeviceToHost);
cudaDeviceSynchronize();

// Comput red map over previous image
red_black_map_overlap<<<1, nMaxThreads, 0>>>(d_pos, d_xs,
                    ceil(pready->h_pos/nMaxThreads), d_previous);
```

The kernel code will turn the pixels defined in the `xs` array into red. As said before, each thread will work on a portion of data long `maxSect` so that the computation load is split.

The initial problem was to find a way to identify the position of the red component given an arbitrary position of any of the three red, green or blue pixels. The image representation of OpenCV `Mat` object is slightly misleading; in fact, the blue component is at position 0 while the red one at position 2. So, an image 3x3 pixels is build in the following way:

B	G	R	B	G	R	B	G	R
B	G	R	B	G	R	B	G	R
B	G	R	B	G	R	B	G	R

Table 18: *Image color component representation, with dimension 3x3*

At this point, we know that the red color component is always int the *third* position and so, given an *absolute* index i, the corresponding red color component position is identified as:

$$red_{position} = i + (2 - i\%3)$$

By applying this idea, the corresponding CUDA kernel is the following one

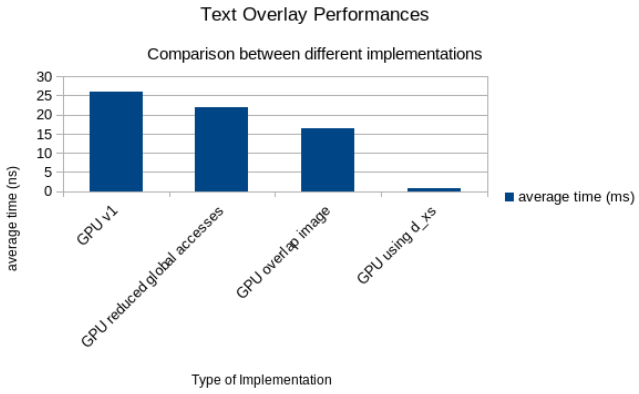
```
__global__ void red_black_map_overlap(unsigned int *pos, int *xs,
                    int maxSect, uint8_t *noise_visualization){
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int start = x * maxSect;
  int max = start + maxSect;

  for (int i = start; i < max && i < *pos; i++) {
    noise_visualization[xs[i] + (2-xs[i]%3)] = 255;
  }
}
```

These are the results from *nvprof*, with `pos` $\simeq$ 70000 (number of changed pixels is around 70000):

Type	**Time** (%)	**Avg**	**Min**	**Max**	**Name**
GPU activities	7.22	915.35us	624.23us	6.7692ms	kernel_red

Table 19: *Profiling result for noise visualizer, overwriting the red pixels over the previous frame using `d_xs` array. The [CUDA memcpy DtoH] and [CUDA memcpy HtoD] are not reported because they can be misleading, since they includes also the cost for the kernel that computes the difference*

By considering only the kernel time, the computation time is reduced by 94.5%, that is pretty impressive.



6 Noise filter

The sensor of an digital camera contains a certain number of pixel, used to acquire the image. The settings of a camera allows to adapt the sensor to the external conditions in order to always get the best shot). In a sunny day, a slow shutter speed, a wide open aperture allows a large number of photons to hit completely the image sensor.

On the other hand, in darker conditions, faster shutter speeds and a closer aperture do not allow enough photons to hit the sensor to generate a suitable voltage to overcome the noise present at the ground. This leads to a low Signal-To-Noise ratio that is directly converted into noise from the internal camera logic. In this case, a correct trim of the of ISO and the shutter aperture time allow to have an image with lower noise; but, in most of the cases, the cheap webcams do not have these advanced configurable feature but perform an auto-adapt procedure.

This is where the *noise filtering* take a major role for this project. Being the data sent based on

the difference with the previous frame, each pixel that is mainly driven by the noise, has higher probability to flip drastically change its value. So, even if the pixel itself does not contains any information, its difference is still sent.

6.1 Convolution filters

Convolution filters are used to perform some perturbation over an image in order to obtain some enhancement. They are based on a *kernel*, a convolution matrix, that is used to perform the convolution between the image and the kernel itself, in the following way:

$$g(x,y) = \sum_{u=\frac{-k}{2}}^{\frac{k}{2}} \sum_{v=\frac{-k}{2}}^{\frac{k}{2}} K[u,v] \cdot I(x-u, y-u)$$

Where K is the kernel and I is the original image; g will be the perturbed resulting image. Depending on the type of selected kernel, the image will be modified in a different way. By defining the size of the kernel as $3 \cdot 3$, the identity is the following:

$$K = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

In this way, the original image is not modified; let's check it:

$$\begin{aligned} I * K &= \left(\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} * \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \right) = \\ &= (i \cdot 0) + (h \cdot 0) + (g \cdot 0) + (f \cdot 0) + (e \cdot 1) + (d \cdot 0) + (c \cdot 0) + (b \cdot 0) + (a \cdot 0) \\ &= e \end{aligned}$$

6.2 Mean filter

The idea of the Mean filter, also called *blur box*, is to set the kernel with the elements that have all the same weight but normalized, like in the following 3x3 convolution matrix:

$$K = \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \begin{bmatrix} 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 \\ 0.11 & 0.11 & 0.11 \end{bmatrix}$$

It can be simple computed in the following way, assuming the kernel is on KxK:

```
float* computeMeanKernel(int K){
  double sum = 0;
  float* k = (float*)malloc(K*K*sizeof(float));
  for (int i = 0; i < K; i++){
    for (int j = 0; j < K; j++){
      k[i*K+j] = 1.0/(K*K);
    }
  }

  return k;
}
```

In this way, the image is basically blurred due to the fact that the pixel at i position is based now also on the values of the surrounding 8 pixels.

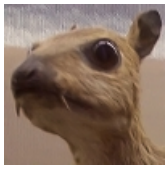


Figure 25: *Original*

Figure 26: *Mean filtering 3x3*

This solution can be exploited in order to make the pixels that suffers from the noise problem more smooth with the respect to the neighbors ones. In this way, even if the two consecutive frames are not exactly the same, the difference among the single pixels will be lower, resorting to a lower data sent over the socket. Let's take for simplicity two frames that are in gray scale, between 0 and 255, that are describes as two 3x3 matrices.

$$A = \frac{1}{9}\begin{bmatrix} 120 & 131 & 112 \\ 112 & 101 & 82 \\ 44 & 106 & 65 \end{bmatrix} \qquad B = \begin{bmatrix} 120 & 139 & 90 \\ 99 & 126 & 106 \\ 46 & 75 & 88 \end{bmatrix}$$

If we simply compute the difference, and we set the threshold to 20, by computing the difference location by location, 5 pixels must be sent as difference. If we apply now the kernel to both the two frames, in order to simulate the behaviour of the algorithm, the two resulting matrices are the following:

$$A = \frac{1}{9}\begin{bmatrix} 51 & 73 & 47 \\ 68 & 96 & 66 \\ 40 & 56 & 39 \end{bmatrix} \qquad B = \begin{bmatrix} 53 & 75 & 51 \\ 67 & 98 & 69 \\ 38 & 60 & 43 \end{bmatrix}$$

As you can see now, no pixels need to be sent. This is a huge improvement, since we have saved 100% of the bandwidth if we neglect the overhead to transmit the information that no pixels have changed. Even if this seems an optimal solution, it has one major drawback. The image will became distorted from the details point of view. In fact, a trade-off between image quality and saved bandwidth must be selected.

6.3 Gaussian filter

The second type of filter that has been selected to reduce the noise, is the *Gaussian filter*, that allows to reach a similar result with the respect to the Mean one. The idea is always to distorce the image to reduce the noise and make the difference among pixels lower. The matrix of a 3x3 Gaussian filter is the following:

$$K = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

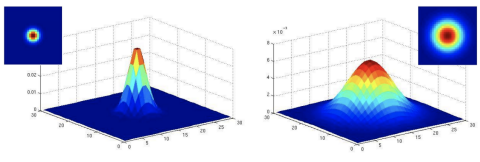
It can be computed using the following formula:

$$K(x, y) = \frac{1}{2 \cdot \pi \cdot \sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

It is called Gaussian filter because of its shape, corresponds to a Gaussian function. A filter of this type requires $6\sigma - 1$ values (to make it sure it's odd), so that for a σ of 3 the kernel must be 17 elements long. The σ value if given is pixels. If we use a square kernel its size can be simply computed as:

$$side_{lenght} = \sqrt{6 \cdot \sigma - 1}$$

Where $side_{lenght}$ is one of the length of the matrix that composes the kernel. A visual example representation is available in the following image.



$\sigma = 2$ with 30×30 kernel

$\sigma = 5$ with 30×30 kernel

Gaussian filter weights pixels into a bell-shape curve around the central pixel. This means that the farther the pixel from the center, the lower the weight. On the other hand the Mean filter is just an average between the center and the neighboring pixels. This means that that for all the pixels, equal weights are assigned, regardless the distance from the center.

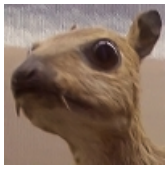


Figure 27: *Original*

Figure 28: *Gaussian filter 3x3*

The C code, that give σ, computes the entire convolution matrix is the following:

```
float* computeGaussianKernel(float s, int K){
  float sum = 0;
  float* k = (float*)malloc(K*K*sizeof(float));
  for (int i = 0; i < K; i++){
    for (int j = 0; j < K; j++){
      double x = i - (K - 1) / 2.0;
      double y = j - (K - 1) / 2.0;
      k[i*K+j] = (1.0/(2.0*M_PI*s*s)) * exp(-((x*x + y*y)/(2.0*s*s)));
      sum += k[i*K+j];
    }
  }
  for (int i = 0; i < K; i++) {
    for (int j = 0; j < K; j++) {
      k[i*K+j] /= sum;
    }
  }

  return k;
}
```

6.4 Tiled CUDA 2D implementation

In order to test the effectiveness of the algorithm and if it is suitable for this project, a configurable implementation has been developed in CUDA. In order to have all the same test environment, two sequential frames have been taken from the webcam in a semi dark environment with a person in the center.

Figure 29: *First frame*

Figure 30: *Second frame*

As you can see, the two images are pretty similar in term of content; a trained eye can see that the head of the second frame is slightly tilted to the right with the respect to the previous one. The idea is to run now the *noise visualizer*, refer to Section 5, on these two frames and understand where the noise is located.

Figure 31: *Red-black representation of the noise*

The noise, is the one in the background that do not contribute in any way to the video flow. In fact, even if it is fixed, it is sent anyway via the socket, increasing the used bandwidth. The CUDA implementation is based on the 2D tiled version but includes the management for all three color components.

The entire algorithm is based on the usage of the shared memory and the synchronization among the thread. The idea is to make each thread work on single pixel and apply the kernel, by using the portion of the image that is in the shared memory, called *tile*.

The kernel implementation is composed of multiple parts:

1. **Shared memory initialization and boundary condition management**: As introduced before, each thread, identified by its ty and tx coordinates should work on a precise pixels, identified as:

```
__shared__ int N_ds[BLOCK_SIZE][BLOCK_SIZE*3];

x = threadIdx.x + blockIdx.x*TILE_SIZE - K/2;
y = threadIdx.y + blockIdx.y*TILE_SIZE - K/2;
```

   In fact, the idea is to create a block of threads that corresponds to `BLOCK_SIZE*BLOCK_SIZE` as dimension, so that each one of the loads into the shared memory the value of its pixel (the multiplication by 3 in the kernel implies that each threads read all three color components). The following code, instead, includes also the write into the shared memory.

```
int tx = threadIdx.x;    //W
int ty = threadIdx.y;    //H
int row_o = blockIdx.y*TILE_SIZE + ty;
int col_o = blockIdx.x*TILE_SIZE + tx;
int row_i = row_o - K/2;
int col_i = col_o - K/2;

if(row_i >= 0 && row_i < H && col_i >= 0 && col_i < W){
  N_ds[ty][tx*3] = image[row_i*W*3+ col_i * C];
  N_ds[ty][tx*3+1] = image[row_i*W*3+ col_i*3 + 1 ];
  N_ds[ty][tx*3+2] = image[row_i*W*3+ col_i*3 + 2 ];
} else {
  N_ds[ty][tx*3] = 0;
  N_ds[ty][tx*3+1] = 0;
  N_ds[ty][tx*3+1] = 0;
}
```

   The value `TILE_SIZE` defines the dimension of the tile, in which each the kernel is applied. The problem is that, the first pixel (position [0,0]) in the image, if the kernel has a dimension of 3x3, need to access also the location [-1,-1], [-1, 0], [-1, 1], [0, -1], [1, -1]. In this case, they are not available. This is the reason of the presence of the if condition, that sets 0 at these locations in the shared memory by "shifting" the indexes by $K/2$. The problem is shown in the below picture:

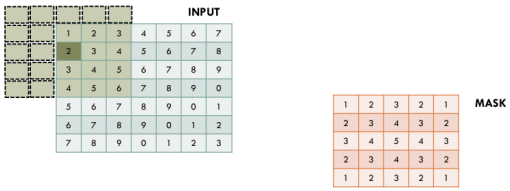


Figure 32: *Boundary conditions among the image border*

   In this way, all the pixels needed to process a tile of `TILE_SIZE`x`TILE_SIZE` dimension, are loaded into the `N_ds` shared memory that is large `BLOCK_SIZE*BLOCK_SIZE*3` in order to contains also the values at the boundary of the tile.

2. **Filter application**: Foreach element in the tile and for all three color components, the kernel is computed using the shared memory.

The entire kernel is the following:

```
__constant__ float dev_k[K*K];

__global__ void convolution_kernel(uint8_t *image, uint8_t *R)
{
  __shared__ int N_ds[BLOCK_SIZE][BLOCK_SIZE*3];

  int tx = threadIdx.x;    //W
  int ty = threadIdx.y;    //H
  int row_o = blockIdx.y*TILE_SIZE + ty;
  int col_o = blockIdx.x*TILE_SIZE + tx;
  int row_i = row_o - K/2;
  int col_i = col_o - K/2;

  // Management of the boudary conditions
  if(row_i >= 0 && row_i < H && col_i >= 0 && col_i < W){
    N_ds[ty][tx*3] = image[row_i*W*3+ col_i * C];
    N_ds[ty][tx*3+1] = image[row_i*W*3+ col_i*3 + 1 ];
    N_ds[ty][tx*3+2] = image[row_i*W*3+ col_i*3 + 2 ];
  } else {
    N_ds[ty][tx*3] = 0;
    N_ds[ty][tx*3+1] = 0;
    N_ds[ty][tx*3+1] = 0;
  }

    __syncthreads();

  if(row_o < H && col_o < W){
    float outputR = 0.0;
    float outputG = 0.0;
    float outputB = 0.0;
    if(ty < TILE_SIZE && tx < TILE_SIZE){
      for(int i = 0; i < K; i++)
      for(int j = 0; j < K; j++){
        outputR += dev_k[i*K+j] * N_ds[i+ty][(j+tx)*3];
        outputG += dev_k[i*K+j] * N_ds[i+ty][(j+tx)*3 + 1];
        outputB += dev_k[i*K+j] * N_ds[i+ty][(j+tx)*3 +2];
      }

      R[row_o*W*3 + col_o*3] = outputR;
      R[row_o*W*3 + col_o*3 + 1] = outputG;
      R[row_o*W*3 + col_o*3 + 2] = outputB;
    }
  }
}
```

The kernel call needs to work on an image of 1920*1080*3 elements divided into square tiles of `TILE_SIZE`x`TILE_SIZE` dimensions. Since the tile needs also the surrounding element to compute the kernel, the block size is composed of `BLOCK_SIZE`x`BLOCK_SIZE` elements, where:

$$BLOCK\_SIZE = TILE\_SIZE + K - 1$$

The grid instead, is configured to generate as many tile as needed from the image. In this case, a grid of `W/TILE_SIZE`x`H/TILE_SIZE` is enough (need to be rounded to the upper value). This is the complete kernel call

```
dim3 blockSize, gridSize;
blockSize.x = BLOCK_SIZE, blockSize.y = BLOCK_SIZE, blockSize.z = 1;
gridSize.x = ceil((float)W/TILE_SIZE),
gridSize.y = ceil((float)H/TILE_SIZE),
gridSize.z = 1;

// Kernel call
convolution_kernel<<<gridSize, blockSize>>>(d_current, d_current_filtered);
```

This solution includes already optimal solution in order to increase the performance and reduce at minimum the overhead introduced by the computation of the filter. In fact, by having defined how convolution if performed, it can be simple called, after have set the kernel in the constant memory.

Having defined the kernel, that once computed is fixed for the entire program execution, allows to not cope with the cache coherence management, allowing data to be aggressively cached in the L1 cache (the nearest to the SM).

Moreover, access to the memory has been optimized by resorting to the usage of the shared memory. Since the larger the dimension of the tile, the higher the ration between the manipulated pixels and the memory accesses, an optimal solution is to increase as much as possible the tile size. The limit is the shared memory, that is limited.

The execution of the kernel is directly proportional to the size of the tile and the kernel dimension; the code shown before has been run for multiple sessions in order to be able to plot the kernel time for multiple convolution matrix and multiple tile size.

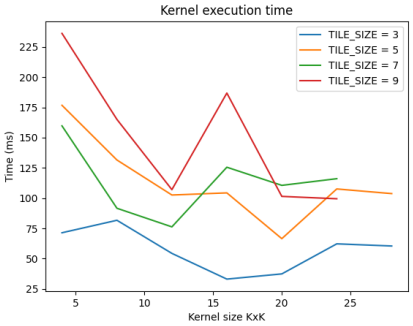
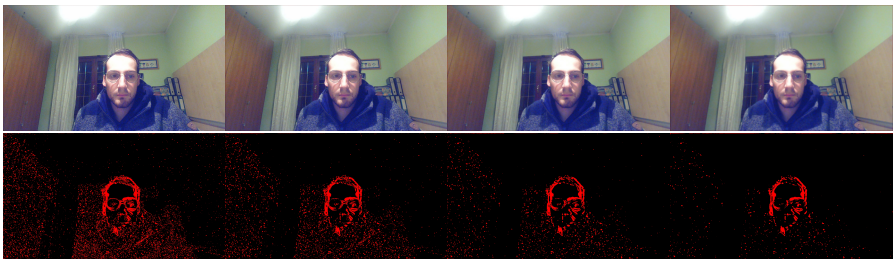


Figure 33: *Kernel execution time from profiler*

Both the Gaussian and the Mean filters have been run over the two frames used as test dataset, the results are available at Table 34. The number of pixels changed, if no filter is applied, is 369350, that is 5.93% of the whole image.

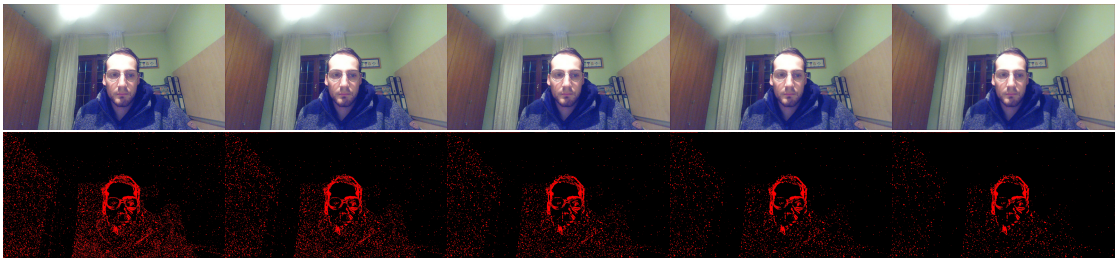
Filter Type	**Kernel size**	**Total saved pixels** (%)
Mean	3	3.37
	5	2.31
	7	1.66
	9	1.24
Gaussian	3 ($\sigma = 1$)	3.58
	4 ($\sigma = 2$)	2.87
	5 ($\sigma = 3$)	2.37
	6 ($\sigma = 5$)	1.98
	7 ($\sigma = 8$)	1.66

Figure 34: *Results of both Gaussian and Mean filter, with different kernel size*



(a) $K = 3$ (b) $K = 5$ (c) $K = 7$ (d) $K = 9$

Figure 35: Mean filtering results



(a) $K = 3$ (b) $K = 4$ (c) $K = 5$ (d) $K = 6$ (e) $K = 7$

Figure 36: Gaussian filtering results

By visually inspecting the results, by using a kernel that is too large, leads to a non-negligible distortion to the image due to the blur effect. By the way, the team has defined that the best visual result is when $K = 4$, using the Gaussian Filter, that allows to avoid to send about 10600 pixels, with the respect to the selected two frames.

6.5 2D CUDA Tiled Median filter

Another algorithm that has been selected to try to remove the noise from the image is the Medial filter. It works in a completely different way from the Mean and Gaussian filters. The idea behind of it is to preserve the useful details of the image while removing the noise.

Like the Mean filter, the Median one takes N neighbor pixels that are next to the selected one but, instead of replacing it with the average value between all of them it replace that pixel with the *median*.

The median of a plan one dimensional array can be computed by first sorting all elements and then take the one in the middle of the array itself. On the other hand, if we have a matrix, the media con be computed in the same way, but all elements are considered and ordered.

1	7	3
2	**9**	6
4	1	6

$\Longrightarrow$ [1, 1, 2, 3, **4**, 6, 6, 7, 9] $\Longrightarrow$ 4

Table 20: *Median of a matrix 3x3*

In this case, the example above shows the media computation of a matrix 3x3 that can corresponds to the kernel size of CUDA kernel.

The CUDA algorithm is structured in the same way as the convolution one, in which the image processing is divided into *tiles* of fixed dimensions. In the same way, pixels are loaded into the shared memory in order to reduce the accesses to the Global memory.

So, the first step to do is to generate a CUDA function that is able to extract the media from a given array, of size $K * K$. The following code shows the implementation of the Bubble Sort and take the median element, by assuming that N is odd:

```
__device__ uint8_t median(uint8_t *array, int N){
  bool swapped = true;
  for (int a = 0; a < N && swapped; a++){
    swapped = false;
    for (int i = 0; i < N - 1; i++){
      if(array[i] > array[i+1]){
        uint8_t tmp = array[i];
        array[i] = array[i+1];
        array[i+1] = tmp;
        swapped = true;
      }
    }
  }

  return array[N/2];
}
```

At this point, the kernel must load into the shared memory the image and then, foreach thread three arrays are computed; one for each of the color components. By iterating of the tile, elements are put in the three arrays and the median extracted. At this point, the new image is computed using that median. The following code shows the kernel for the media filtering.

```
__global__ void median_kernel(uint8_t *image, uint8_t *R)
{
  __shared__ uint8_t N_ds[BLOCK_SIZE][BLOCK_SIZE*3];

  int tx = threadIdx.x;    //W
  int ty = threadIdx.y;    //H
  int row_o = blockIdx.y*TILE_SIZE + ty;
  int col_o = blockIdx.x*TILE_SIZE + tx;
  int row_i = row_o - K/2;
  int col_i = col_o - K/2;

  if(row_i >= 0 && row_i < H && col_i >= 0 && col_i < W){
    N_ds[ty][tx*3] = image[row_i*W*3+ col_i * C];
    N_ds[ty][tx*3+1] = image[row_i*W*3+ col_i*3 + 1 ];
    N_ds[ty][tx*3+2] = image[row_i*W*3+ col_i*3 + 2 ];
  } else {
    N_ds[ty][tx*3] = 0;
    N_ds[ty][tx*3+1] = 0;
    N_ds[ty][tx*3+1] = 0;
  }

  __syncthreads();

  if(row_o < H && col_o < W && ty < TILE_SIZE && tx < TILE_SIZE){
    uint8_t medR[K*K];
    uint8_t medG[K*K];
    uint8_t medB[K*K];
    int r = 0;
    int g = 0;
    int b = 0;

    // Copying elements into the three arrays
    for(int i = 0; i < K; i++)
      for(int j = 0; j < K; j++){
        medR[r++] = N_ds[i+ty][(j+tx)*3];
        medG[g++] = N_ds[i+ty][(j+tx)*3 + 1];
        medB[b++] = N_ds[i+ty][(j+tx)*3 + 2];
      }

    // Computation of the median
    R[row_o*W*3 + col_o*3] = median(medR, K*K);
    R[row_o*W*3 + col_o*3 + 1] = median(medG, K*K);
    R[row_o*W*3 + col_o*3 + 2] = median(medB, K*K);
  }
}
```

The problem with this implementation is that, it needs to sort an array of $K * K$ elements for each color component in the image, that obviously implies a longer computation. The below images are from the execution of the above algorithm, using $K = 5$ and a frame of 1920*1080*3 elements. The kernel execution time from the *profiler* is 574.67ms, that is a lot if we want to keep the FPS sufficiently high.

Figure 37: *Median filtered frame*

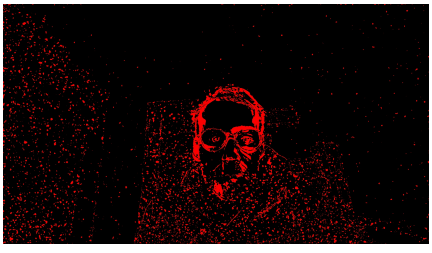


Figure 38: *Red-black noise visualization*

Figure 39: *Example of Median filter with* $K = 5$

Let's suppose for a moment that we do not have any constraint from the FPS; *is it a good solution for the noise?* From the point of view of the percentage of the total pixels saved, we have only 2.54%, that is even higher the result from a simpler Mean filter with the same kernel size.

The problem with this filter is comping from the type of noise that is characterizing out frames. In fact, the Median filter works better on *Salt and Pepper* noise, that are some random pixels that have completely different values from the point of view of the magnitude. Let's see an example:

1	7	3
2	**89**	6
4	1	6

$\implies$ [1, 1, 2, 3, **4**, 6, 6, 7, 89] $\implies$ 4

Table 21: *Median of a matrix 3x3, with salt and pepper noise*

Figure 40: *Salt and pepper noise*

So, due to the fact that the Median filter does not bring any advantage, it has not been optimized and even used. Beside this, the implementation allowed us to better understand the nature of the noise.

7 Grayscale Filter

In digital photography, computer-generated imagery, and colorimetry, a grayscale image is one in which the value of each pixel is a single sample representing only an amount of light; that is, it carries only intensity information. Grayscale images, a kind of black-and-white or gray monochrome, are composed exclusively of shades of gray. The contrast ranges from black at the weakest intensity to white at the strongest. Grayscale images (such as photographs) intended for visual display (both on screen and printed) are commonly stored with 8 bits per sampled pixel. In fact the range of the single pixel goes from 0 (black), to 255 (white). There are 2 different methods to convert an image from RGB to Grayscale. As it's widely known, an RGB image is made of three channels of 8 bits each. The algorithm need to convert the three channels into a signle one, and so the straightforeword implementation is based on the following mathematical formula:

$$grayscale[i] = \frac{R[i] + G[i] + B[i]}{3}$$

where i represent the i-esim pixel of the image. This method, because is executing an average of three values, is called `Average Method`. However the average method is simple but doesn't work as well as expected. The reason being that human eyeballs react differently to RGB. Eyes are most sensitive to green light, less sensitive to red light, and the least sensitive to blue light. Therefore, the three colors should have different weights in the distribution. Knowing this we introduce the `Weighted Method`.

The weighted method, also called luminosity method (which are used in standard color TV and video systems such as PAL, SECAM, and NTSC), weighs red, green and blue according to their wavelengths. The improved formula is as follows:

$$grayscale[i] = 0.299 \cdot R[i] + 0.587 \cdot G[i] + 0.114 \cdot B[i]$$

7.1 CPU Implementation

In the project the algorithm is slightly modified. In fact there is the need of keeping the frame on 3 channels. However a single channel from 0 to 255 is the same as having three channels from 0 to 255 with the **same value**. Knowing the algorithm behid the filter, the implementation is pretty easy. In fact it is simply a for loop that iterates over all pixel of the matrix. The specific code is the following:

```
for (int i = 0; i < total; i = i + 3) {
  int sum = ready.data[i] +ready.data[i+1] + ready.data[i+2];
  ready.data[i] = sum/3;
  ready.data[i+1] = sum/3;
  ready.data[i+2] = sum/3;
}

```

By viewing the code, it's easy to understand that the sum is computed and then the data is changed in all of the three channels. This implementation is the easiest and the most straightforeword, and for this reason it won't be the faster.

The results are shown in Figure 41

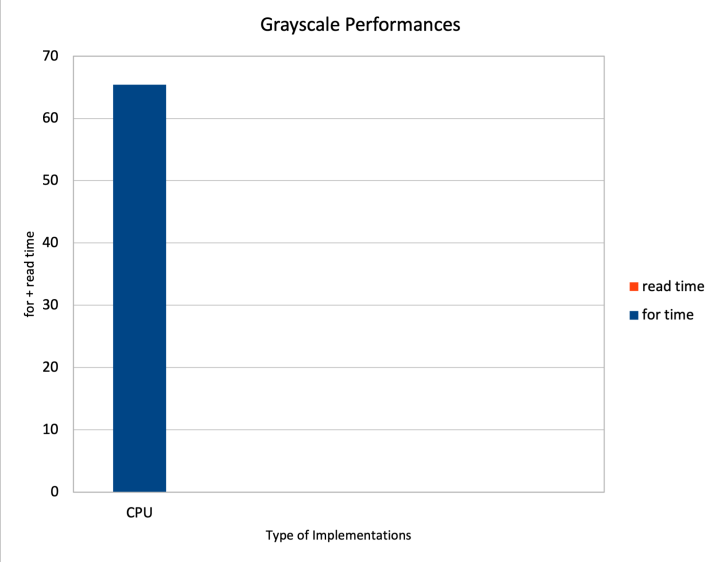


Figure 41: *CPU results*

7.2 GPU First Version

For the first version, threads are organized as explained in subsection 2.2. For what concerns the memory allocation, obviously the CPU address space and the GPU one are two separated things, so a specific allocation on the device side must be done. This can be accomplished thanks to some CUDA's API as follows:

```
uint8_t *d_current, *grayscale;

cudaMalloc((void **)&d_current, total * sizeof *d_current);
cudaMalloc((void **)&d_grayscale, total * sizeof *d_grayscale);

```

After this step is important to copy the data from the CPU (Host) to the GPU (Device). In fact the following operation is executed:

```
cudaMemcpy(d_current, frameData, total, cudaMemcpyHostToDevice);

```

After that the data are ready and so the kernel can be launched.

```
grayscale_kernel_v2<<<1, nMaxThreads>>>(d_current, d_grayscale, maxAtTime);

```

After that, the results can be copied back from the device to the host, in order to show them to the user. To achieve this goal, the following line of code is executed.

```
1  cudaMemcpy(showReadyNData, d_grayscale, total, cudaMemcpyDeviceToHost);
2
```

The current frame and the grayscale one can be seen in Figure 42 and Figure 43

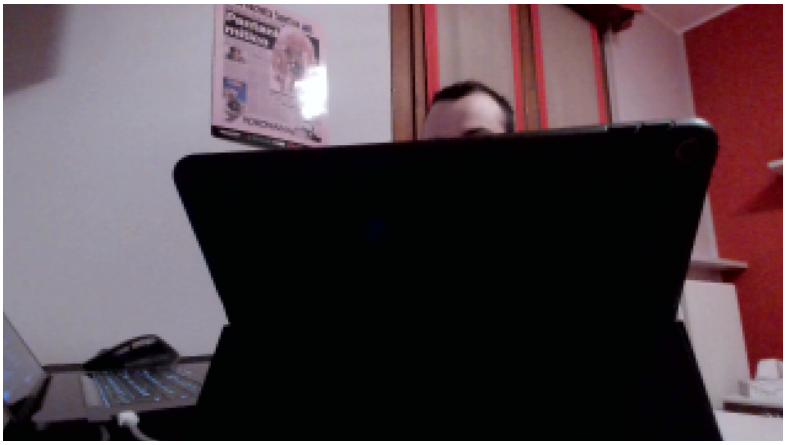


Figure 42: *Frame with colors*

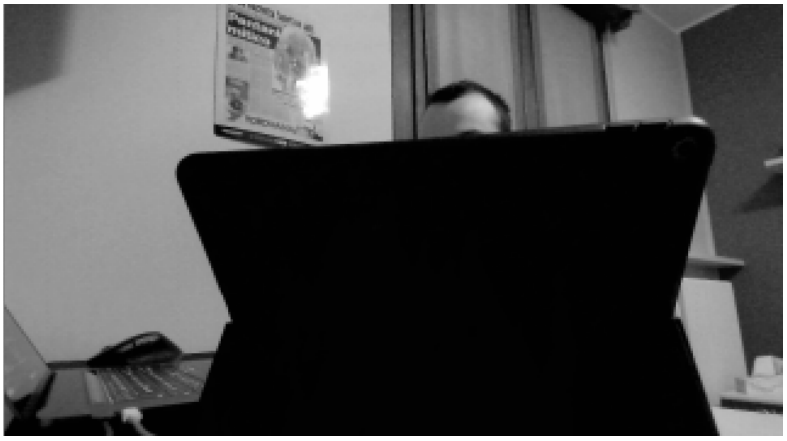


Figure 43: *Frame in black and white*

7.3 GPU Improved Version

As explained in subsubsection 2.2.2 CUDA offers a variant of the APIs that are asynchronous with respect to the device execution. This means that when one of these APIs is called, the operation starts on the device side but the control returns immediately to the host despite the end of the operation on the GPU.

The nice thing about this APIs, this solution is slightly more efficient because is possible to assign to the GPU all the jobs in a consecutive way then waiting instead of assign a job at a time.

The code becomes the following:

```
1  cudaMemcpyAsync(d_current, frameData, total, cudaMemcpyHostToDevice);
2  grayscale_kernel_v2<<<1, nMaxThreads>>>(d_current, d_grayscale, maxAtTime);
3  cudaMemcpyAsync(showReadyNData, d_grayscale, total, cudaMemcpyDeviceToHost);
4
```

By using the GPU, as it's represented in Figure 44, the performance improve a lot.

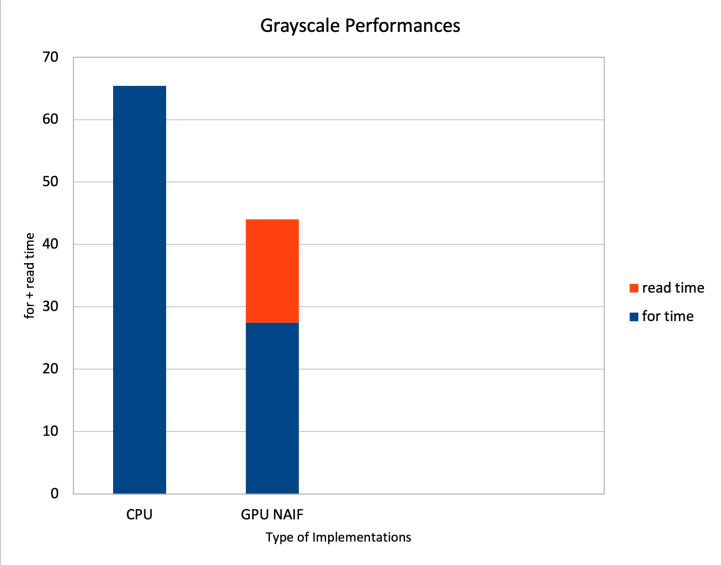


Figure 44: *Frame with colors*

7.4 GPU Global Memory Acess

A very big problem of the current version of the kernel is the very high number of access in the global memory. This causes a very high overhead, because accessing global memory is the real bottleneck. To improve performace, the idea is to increase the size of each read or write in memory. This concept is well explained in subsubsection 2.2.4.

The kernel code now becomes the following:

```
1  __global__ void grayscale_kernel_v2(uint8_t *color, uint8_t *grayscale, int maxSect) {
2    int x = threadIdx.x + blockDim.x * blockIdx.x;
3    int start = x * maxSect;
4    int max = start + maxSect;
5    chunk_t cc;
6    int size = sizeof(chunk_t);
7    int sum = 0;
8
9    for (int i = start; i < max; i++) {
10     cc = ((chunk_t *)color)[i];
11     for (int j = 0; j < size; j++) {
12       sum += ((uint8_t *)&cc)[j];
13       if (((i * size) + j) % 3 == 2) {
14         grayscale[(i * size) + j] = sum / 3;
15         grayscale[(i * size) + j - 1] = sum / 3;
16         grayscale[(i * size) + j - 2] = sum / 3;
17         sum = 0;
18       }
19     }
20   }
21 }
22
```

The implementation of this feature improves performance a little bit, as shown in Figure 45

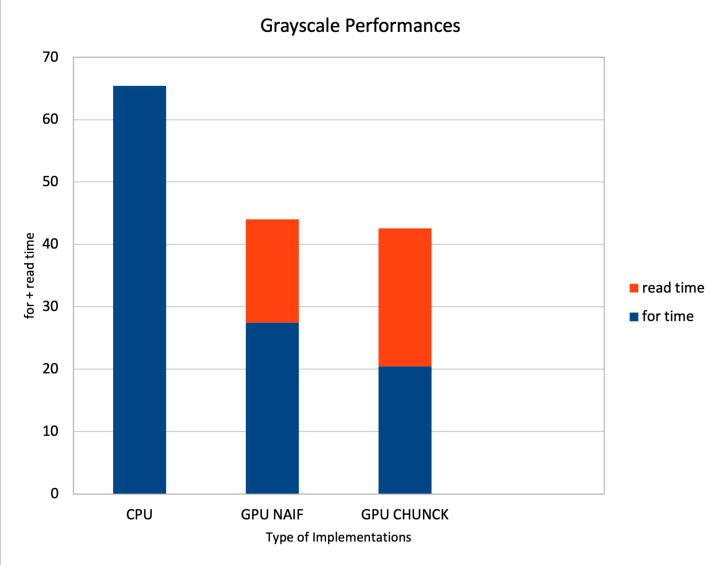


Figure 45: *Frame with colors*

7.5 GPU weighted average

The last improvements is not performance but feature oriented. Before passing to the real code implementation is important to have a little bit of background of how the human eyes are made.

7.5.1 Physiology of the eye

Eyes are organs of vision that detect light. Some eyes just detect whether the surroundings are light or dark while more complex eyes can distinguish shapes and colors. In humans and most other vertebrates, the eye allows light to enter and projects it onto the retina at the rear of the eye. The retina consists of photoreceptor cells called rods and cones. The cones are at the center of the retina, and the rods are at the outer edge of the retina. There are about seven million cones and a hundred million rods. At the retina, the light is detected and converted to electrical signals. These signals are then transferred to the brain through the optic nerve. The brain converts these signals into our view of the world.

The human eye detects visible light, part of the electromagnetic spectrum. Light is produced by rearranging electrons in atoms and molecules. The wavelengths of visible light range from about 400 nm to about 700 nm. Photopic vision is the scientific term for human vision under normal lighting conditions during the day. This allows for color perception. The cones in the retina help us to see the colors. There are three types of cones in the human eye. Each type of cone absorbs light waves of specific frequencies: long wavelengths (L), medium wavelengths (M), and short wavelengths (S). The peak wavelength of L is 564 nm, yellowish-green. The peak of M is 534 nm, bluish-green. The peak of S is 420 nm, bluish-violet. This is shown in Table 22.

Type of cone	Name	Field	Peak of wavelength
S	β	400 nm - 500 nm	420 nm - 440 nm
S	γ	450 nm - 630 nm	535 nm - 555 nm
S	ρ	500 nm - 700 nm	565 nm - 580 nm

Table 22: *Table of the wavelength of the cone of the retina*

Colors are perceived when the cones are stimulated. The color perceived depends on how much each type of cone is stimulated. Yellow is perceived when the yellow-green receptor is stimulated slightly more than the blue-green receptor. The eye is most sensitive to green light (555 nm) because green stimulates two of the three kinds of cones, L and M, almost equally. This is clearly visible in Figure 46.

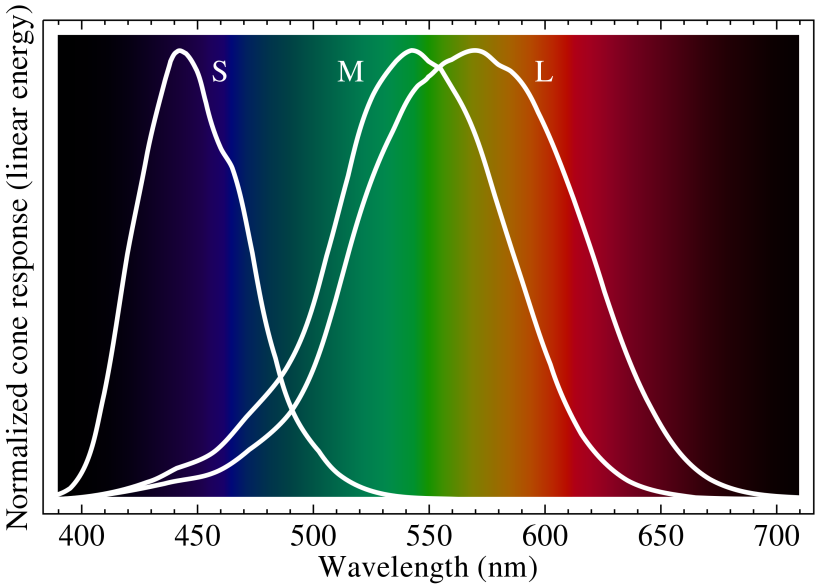


Figure 46: *Cone response*

7.5.2 GPU implementation

Reading subsubsection 7.5.1 it may seem very difficult to implement all this little details into a programming language code. However, as shown in the formula in section 7, it simply is a weighted sum. The code of the kernel becomes the following:

```
__global__ void grayscale_kernel_v2(uint8_t *color, uint8_t *grayscale,
  int maxSect) {
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int start = x * maxSect;
  int max = start + maxSect;
  chunk_t cc;
  int size = sizeof(chunk_t);
  float sum = 0;

  for (int i = start; i < max; i++) {
    cc = ((chunk_t *)color)[i];
    for (int j = 0; j < size; j++) {
      // B
      if (((i * size) + j) % 3 == 0) {
        sum += 0.114 * ((uint8_t *)&cc)[j];
      }
      // G
      if (((i * size) + j) % 3 == 1) {
        sum += 0.587 * ((uint8_t *)&cc)[j];
      }
      if (((i * size) + j) % 3 == 2) {
        sum += 0.299 * ((uint8_t *)&cc)[j];
        grayscale[(i * size) + j] = (uint8_t)sum;
        grayscale[(i * size) + j - 1] = (uint8_t)sum;
        grayscale[(i * size) + j - 2] = (uint8_t)sum;
        sum = 0;
      }
    }
  }
}

```

With this modification we don't obtain any kind of performance improvement, however there is an important scientific improvement. We even expect a performance deterioration, because the code inside the kernel is made of more lines compared to the previous one. This is shown in Figure 47

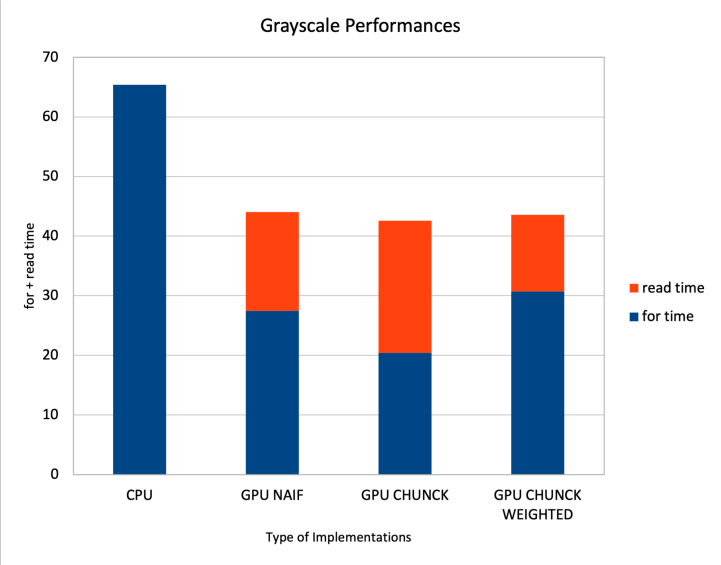
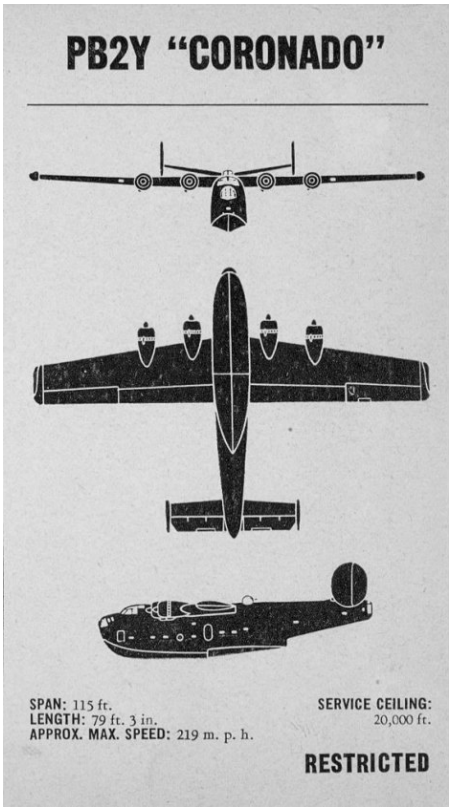


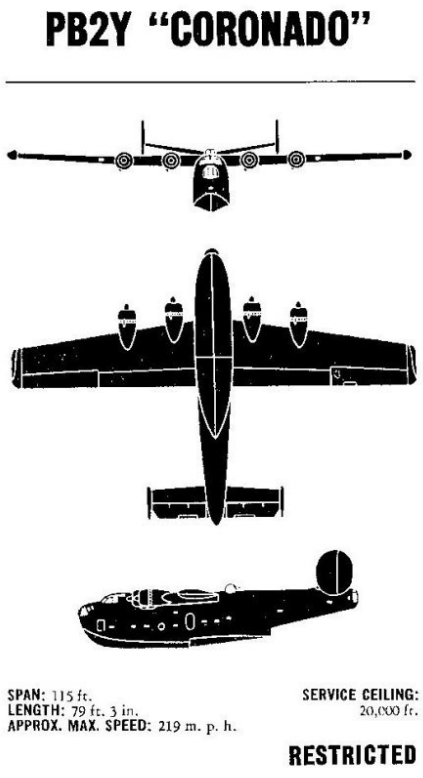
Figure 47: *GPU last version performance*

8 Binarization

Image binarization is the process of taking a grayscale image and converting it to black-and-white, essentially reducing the information contained within the image from 256 shades of gray to 2: black and white, a binary image. It enhances the performance of document processing techniques like OCR and layout analysis. Another domain of application is the segmentation of an image into foreground and background. An example is shown in Figure 48a and Figure 48b.



(a) *Black and White Images*



(b) *Binarized Images*

Figure 48: A figure with two subfigures

The key point is: **"How do we decide from which value the pixel become black (0) or white (255)**. So is important to select a threshold, like shown in Figure 49.

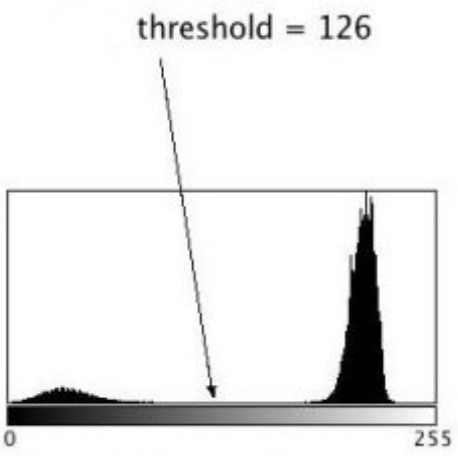


Figure 49: *Distribution of the grayscale pixels*

Let's go step by step and explain what are the main point to exploit in order to obtain a binarize picture.

1. Convert color to grayscale
2. Generate histogram of the grayscale image
3. Compute the two max of the histogram
4. Compute the threshold starting from the two max of the histogram
5. Convert the grayscale to binary knowing the threshold

8.1 Convert color to grayscale

This is the first step, but we won't go into details, because everything has alredy been explained in section 7.

8.2 Generate histogram of the grayscale image

In this step the histogram of the grayscale image is generated. Wehn talking about histogram, the idea is to count the occurencies of the numbers from 0 to 255. In fact, after converting to grayscale, we know that the matrix belonging to the image will have value between 0 and 255, like shown in Table 23

0	0	0	255	255	255	0	0	0
255	255	255	130	130	130	255	255	255
255	255	255	10	10	10	255	255	255

Table 23: *Matrix of a grayscale image*

The histogram has to count the number of occurency of the numbers between 0 and 255, like shown in

Position	0	...	10	...	...	130	...	255
Occurency	2	...	1	...	...	1	...	5

Table 24: *Histogram distribution*

In order to implement everything in code, the idea is to iterate over the full matrix (jumping from 0 to 3, to 6, to 9 etc, in order to exploit the three channels), and count the number of occurencies.

8.2.1 CPU implementation

The CPU implementtion is as simply as it may look: a for loop that iterates over all pixels of the matrix and at the position *histogram[index]* increments the value, where *index* is the value, from 0 to 255, of the i-esim pixel of the image. This is the code:

```
for (int row = 0; row < H; row++) {
  for (int col = 0; col < W; col++) {
    int index = bw.at<uchar>(row, col);
    histogram[index]++;
  }
}

```

8.2.2 GPU Naive implementation

Going from the CPU to GPU is as simple as copying the same code to a GPU kernel and correctly position the thread index. This is the kernel:

```
__global__ void generate_histogram(uint8_t *grayscale, int *histogram,
  int maxSect) {
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int start = x * maxSect;
  int max = start + maxSect;

  for (int i = start; i < max; i = i + 3) {
    grayscale[i]++;
  }
}

```

This code is **Completely Wrong**. In fact there is a race condition, where many thread try to access the same memory cell of grayscale, and for this reason the histogram get generated with less value that what is expected.

The problem is that the operation *grayscale[i]++* is not atomic, so not executed in one clock cycle). To correct this code, the operation is changed to *atomicAdd(&histogram[grayscale[i]], 1)*. So the final code becomes:

```
__global__ void generate_histogram(uint8_t *grayscale, int *histogram,
  int maxSect) {
  int x = threadIdx.x + blockDim.x * blockIdx.x;
  int start = x * maxSect;
  int max = start + maxSect;

  for (int i = start; i < max; i = i + 3) {
      atomicAdd(&histogram[grayscale[i]], 1);
  }
}

```

With the following correct, the histogram is generated correctly.

8.2.3 GPU Shared Memory

In order to improve performance, the idea is to use the shared memory. Every block will write to the shared memory and, only when all threads have finished, the data is added to the vector in the global memory. This is the code:

```
__global__ void generate_histogram_v2(uint8_t *grayscale, int *histogram,
  int maxSect) {
  int tid = threadIdx.x + blockDim.x * blockIdx.x;
  int start = tid * maxSect;
  int max = start + maxSect;

```

```
7       __shared__ int shared_histogram[256];
8
9       if (tid < 256){
10        shared_histogram[tid] = 0;
11      }
12
13      __syncthreads();
14
15      for (int i = start; i < max; i = i + 3) {
16        atomicAdd(&shared_histogram[grayscale[i]], 1);
17      }
18
19      __syncthreads();
20
21      if (tid < 256){
22        atomicAdd(&histogram[tid], shared_histogram[tid]);
23      }
24
25    }
26
```

8.3 Compute the two max of the histogram

After having computed the histogram is important to find a way to compute the threshold. There are two implementation, a simple one and a difficult one:

1. Find the two max: this is the easy way. In fact there is the need to find the two maximum value and make an average.

2. Find the two peak: this is the more complex version. In fact, it would be easy if the value would be fully increasing or fully deacrising. However, in out case, we may have a mix of ascending and descending value, so it gets very complicated.

In this project a simple version is implemented. As a future work, the more difficult (and accurate) version may be implemented.

The algorithm to compute two max of an array in CUDA is based on the reduction algorithm. The schematic of the algorithm is shown in Figure 50 and Figure 51.

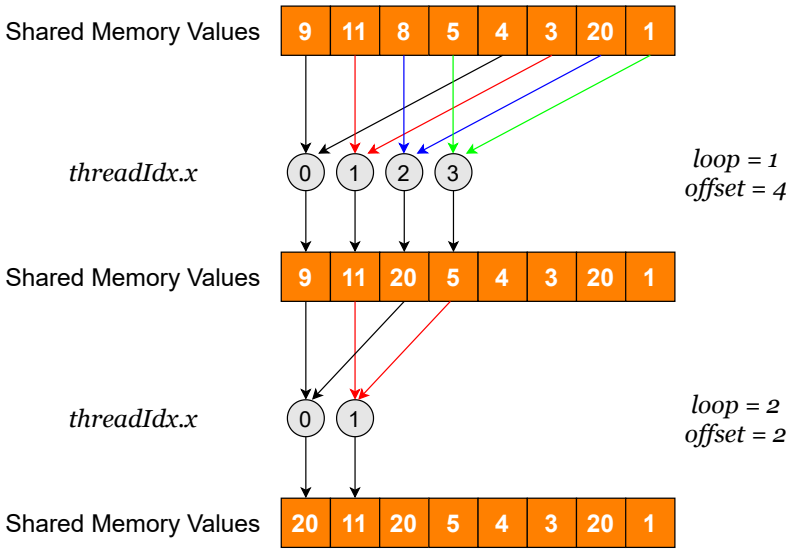


Figure 50: *Reduction algorithm values*

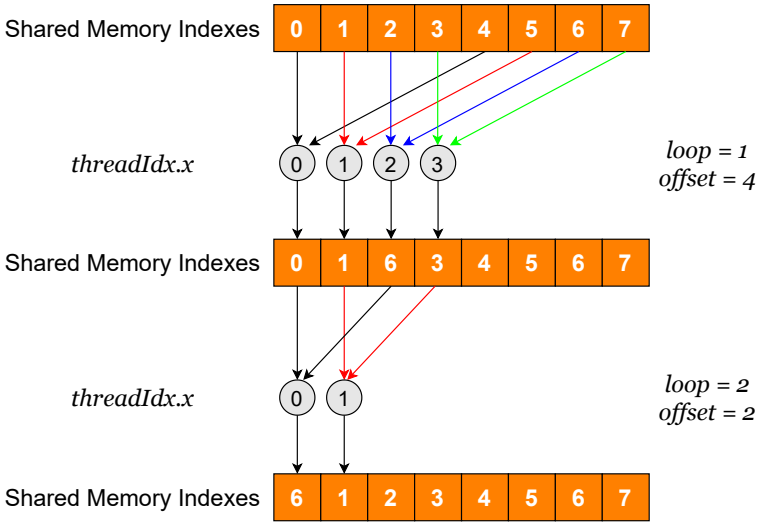


Figure 51: *Reduction algorithm indexes*

The reduction algorithm needs a number of thread that is the same as the numbers of the vector, because initially is important to set the value inside the shared memory. The following code is the one used to declared the shared memory and initialized its values. After that, in order to be sure that all threads have initialized the shared memory, it is important to sync all threads.

```
__global__ void compute_max(int *histogram, uint8_t *indexes_max) {
  int tid = threadIdx.x + blockDim.x * blockIdx.x;

  __shared__ int shared_histogram[256];
  __shared__ int shared_indexes[256];

  shared_histogram[tid] = histogram[tid];
  shared_indexes[tid] = tid;
  __syncthreads();

  // Other operations
}

```

Now let's analyse the core algorithm. Supposing we call the kernel with a single block that contains *N_values* number of threads, where *N_values* is the number of values inside the array, that in out case is 256 (because we go from 0 to 255). The logic behind the algorithm is that in the first iteration, threads needs to access index i of the vector and index i + *blockDim.x / 2*. The operation to execute is to check which value is greater, and then write it to the shared memory. The same is done for the index, in order to retrieve at which indexes are the max values. At the end of the iteration is important to sync all threads, in order to wait that the shared memory has correctly been written. For the following iteration, is necessary to divide by 2 the offset and continue like this until the offset is > 1. In fact, with this condition, the algorithm stops when the offset is equal to 2.

```
  for (unsigned int offset = blockDim.x / 2; offset > 1; offset >>= 1) {
    if (threadIdx.x < offset) {
      if (shared_histogram[threadIdx.x] < shared_histogram[threadIdx.x +
        offset]) {
        shared_histogram[threadIdx.x] = shared_histogram[threadIdx.x +
        offset];
        shared_indexes[threadIdx.x] = shared_indexes[threadIdx.x + offset];
      }
    }
    // sync threads required to ensure all threads have finished writing
    __syncthreads();
  }

```

The last operation is copying back the data from the shared memory to the device data. Since is important to retrieve the two indexes of the two max values, they will be positioned at index 0

and 1 of the shared memory.

```
  if (tid == 0) {
    indexes_max[tid] = shared_indexes[tid];
  }
  if (tid == 1) {
    indexes_max[tid] = shared_indexes[tid];
  }

```

8.4 Compute the threshold

After having computed the two max, the evaluation of the threshold is pretty straightforeword. The formula is the following:

$$threshold = \frac{max1 + max2}{2}$$

An important detail is that if the threshold is too low it means that the the distribution is confined to small values. For this reason, if the threshold is smaller than 50, the threshold will be set manually to 50, in order to correctly detect the background. For the same reason, if the threshold if higher that 200 it will the set manually to 200.

8.5 Convert the grayscale to binary

Knowing the correct threshold, the conversion from grayscale to binary is very easy. The pseudocode is the following;

```
  for (int i = 0; i < total_pixels; i = i + 3) {
    if(matrix[i] > threshold){
      matrix[i] = 255;
      matrix[i+1] = 255;
      matrix[i+2] = 255;
    } else {
      matrix[i] = 0;
      matrix[i+1] = 0;
      matrix[i+2] = 0;
    }
  }

```

The loop is incremented 3 by 3, because as alredy explained, there are 3 channels but, because they all have the same value, there is not the need to check all 3 channles. However, when modifying the channels, all of them need to be changed.