Politecnico di Torino
GPU Programming

REPORT: PROJECT WEBCAM STREAMING & FILTERS

December 29, 2021

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.

$$\begin{tikzcd}
t_1 \arrow[rr, bend left] & & t_2 \arrow[dl] \\
& t_3 \arrow[ul] &
\end{tikzcd}$$

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:

```
int total = 3 * ctx.sampleMat->rows * ctx.sampleMat->cols; // no. bytes
Mat pvs = pready->pframe->clone();
pready->h_pos = 0;

for (int i = 0; i < total; i++) {
  int df = pready->pframe->data[i] - previous.data[i];
  if (df < -LR_THRESHOLDS || df > LR_THRESHOLDS) {
    pready->pframe->data[pready->h_pos] = df;
    pready->h_xs[pready->h_pos] = i;
    pready->h_pos++;
  } else {
    pvs.data[i] -= df;
  }
}

previous = pvs;
```

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.

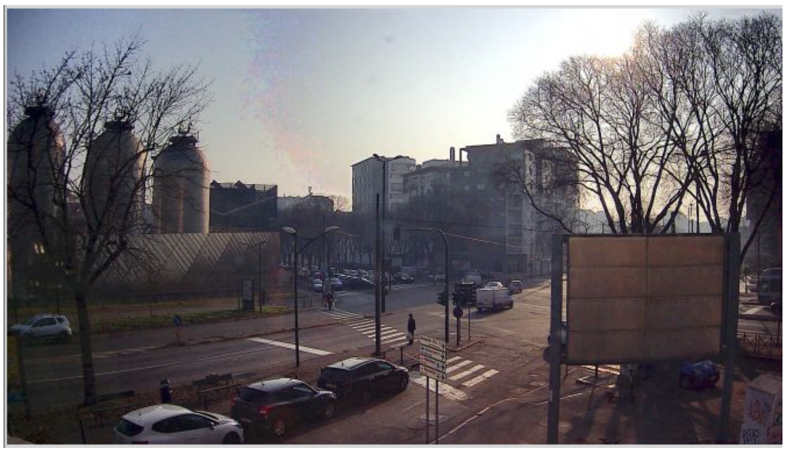

Figure 1: *The result of the reconstructed frame*

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.

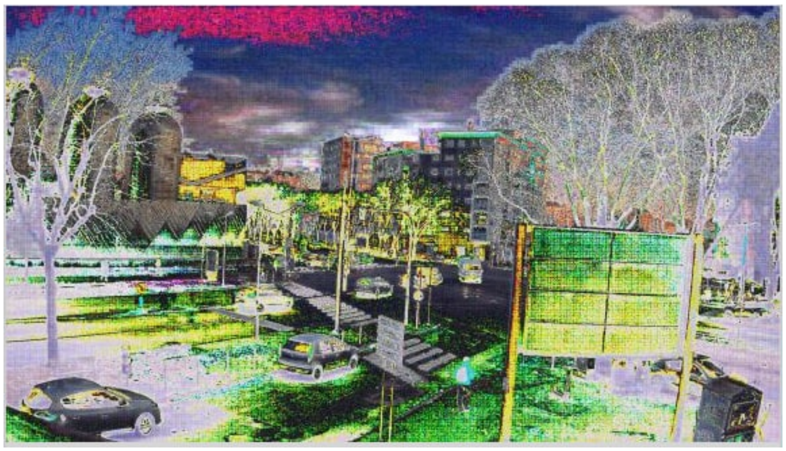

Figure 2: *What happens after a certain time if the error is not take under consideration*

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.

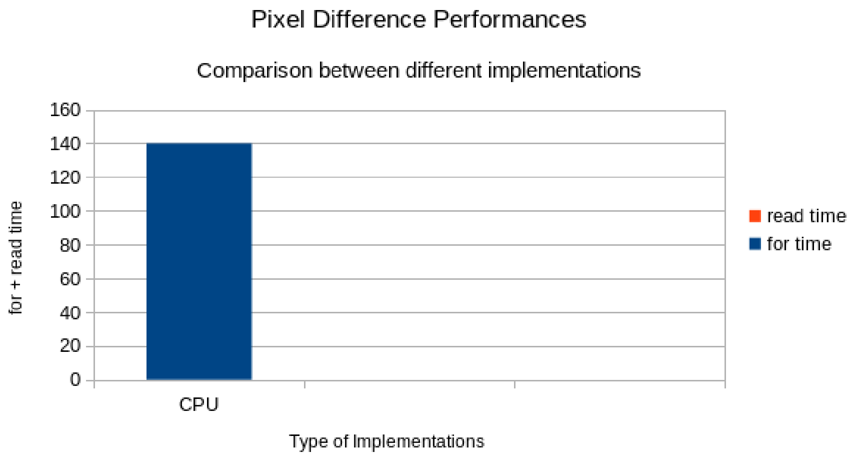

Figure 3: *Pixel Difference Comparison*

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	0	0	0	0	0	0	0	0
0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8

Table 1: *Row major rapresentation in memory*

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;


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

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

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

  }

}

```

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 concerning 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.

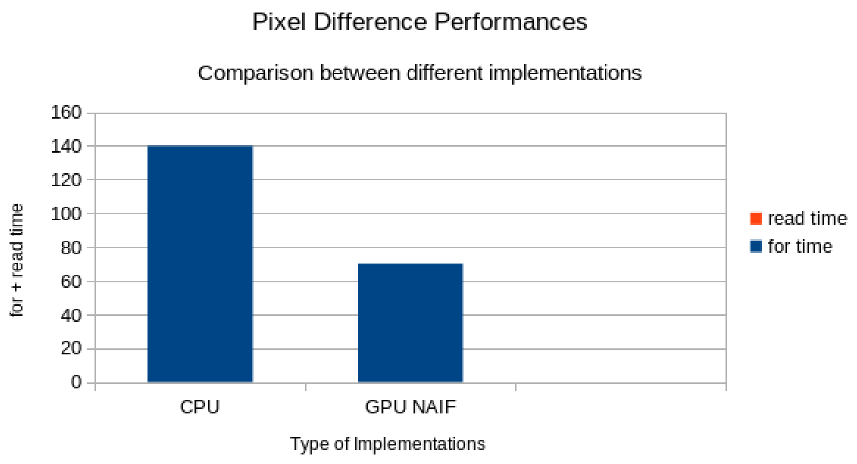

Figure 4: *Pixel Difference Comparison*

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. However 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:

```
__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;

  // counter reset done by thread with index = 0
  if (!x) *pos = 0;
  __syncthreads();


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

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

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

  }

}
```

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.

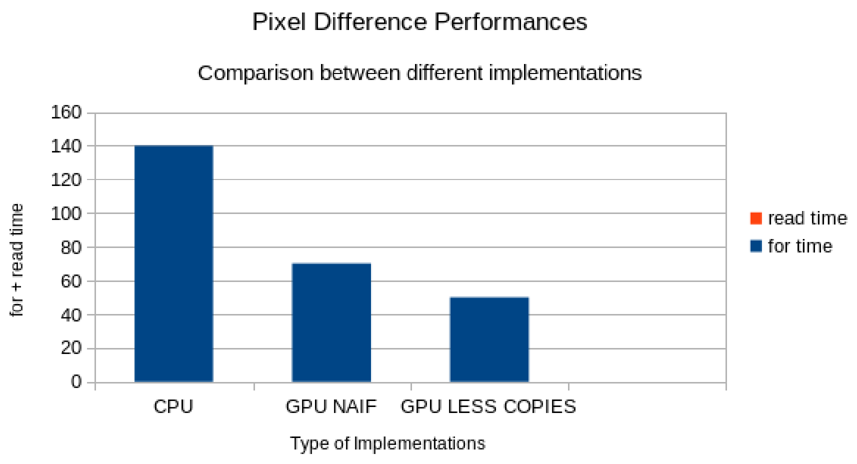

Figure 5: *Pixel Difference Comparison*

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.

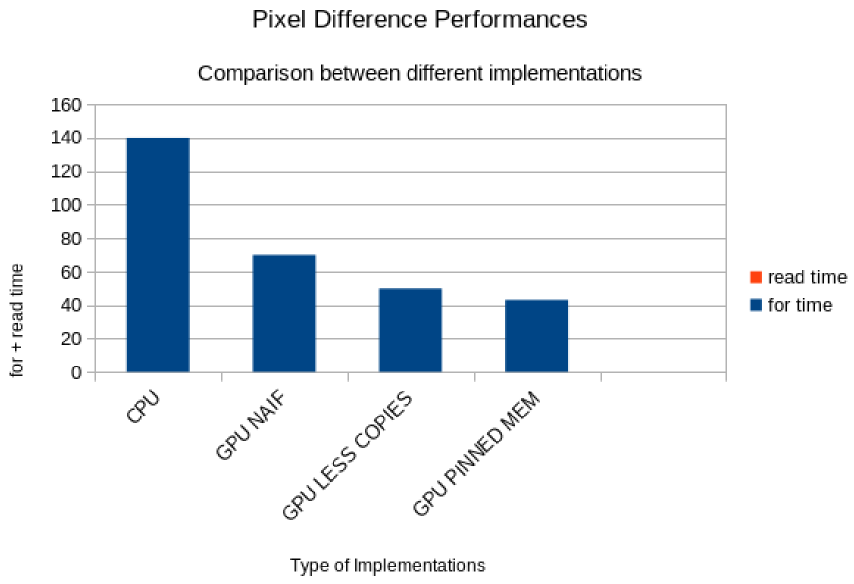

Figure 6: *Pixel Difference Comparison*

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 the case of a low probability of two writes at the same time, 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

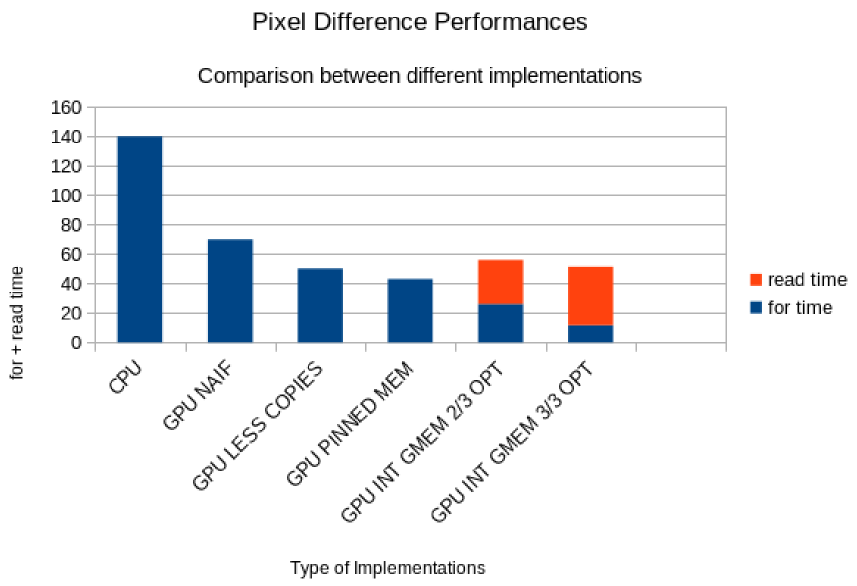

Figure 7: *Pixel Difference Comparison*

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.

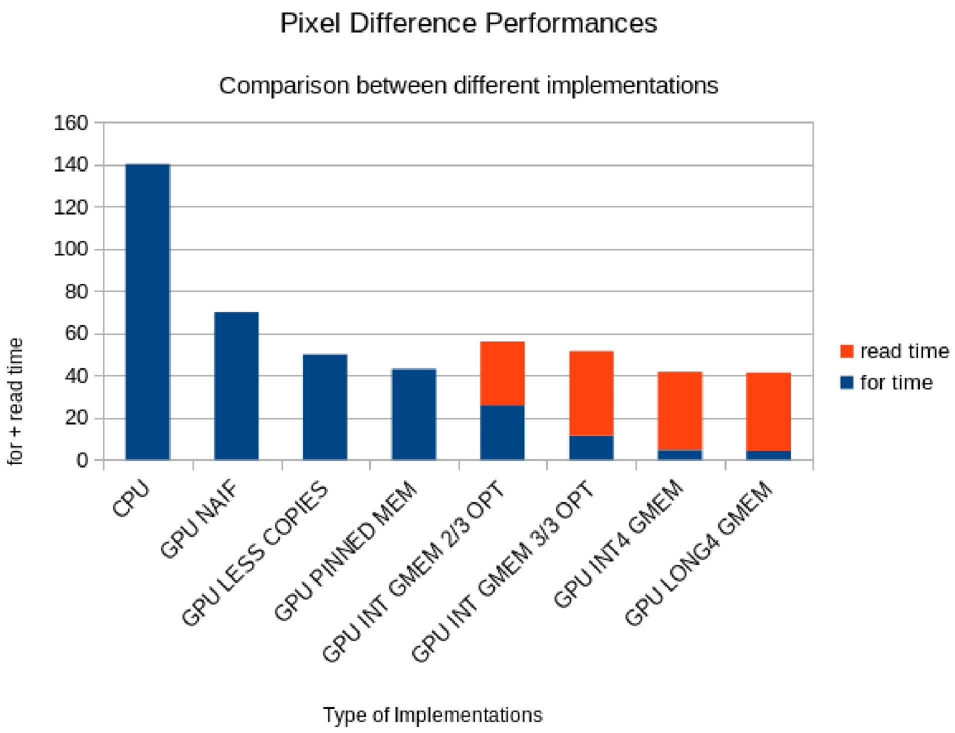

Figure 8: *Pixel Difference Comparison*

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.

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 send 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:

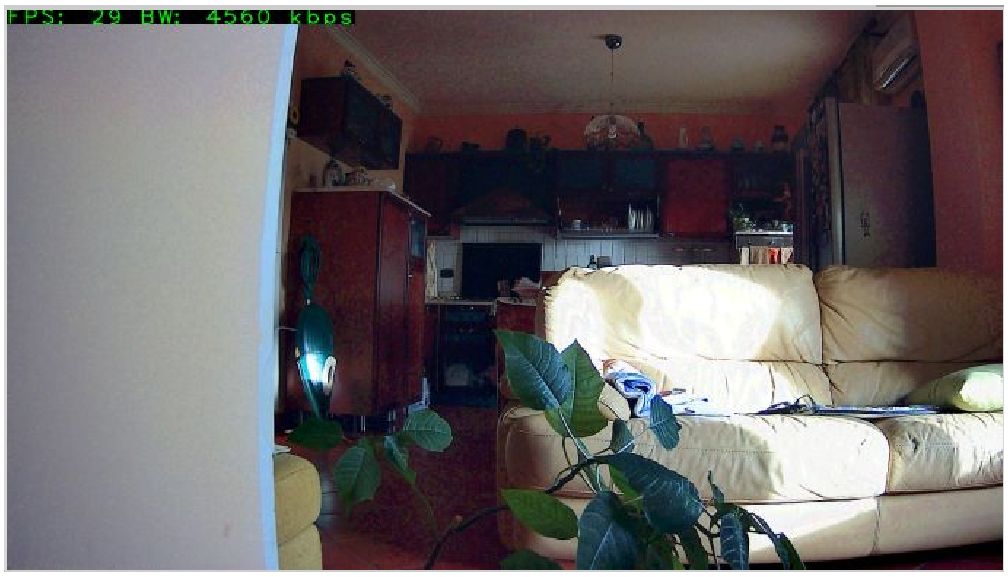

Figure 9: *Text Overlay*

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:

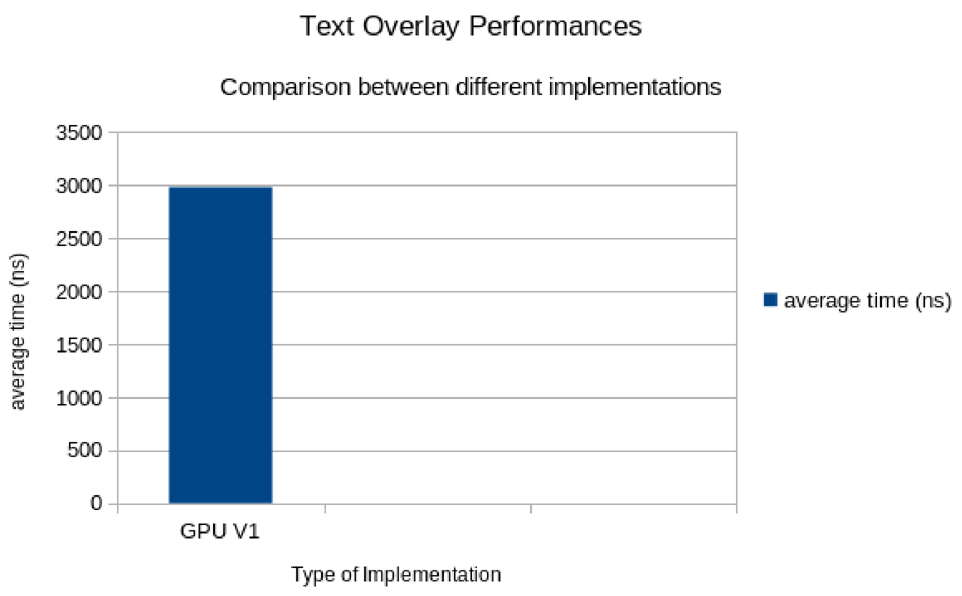

Figure 10: *Text Overlay Comparison*

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:

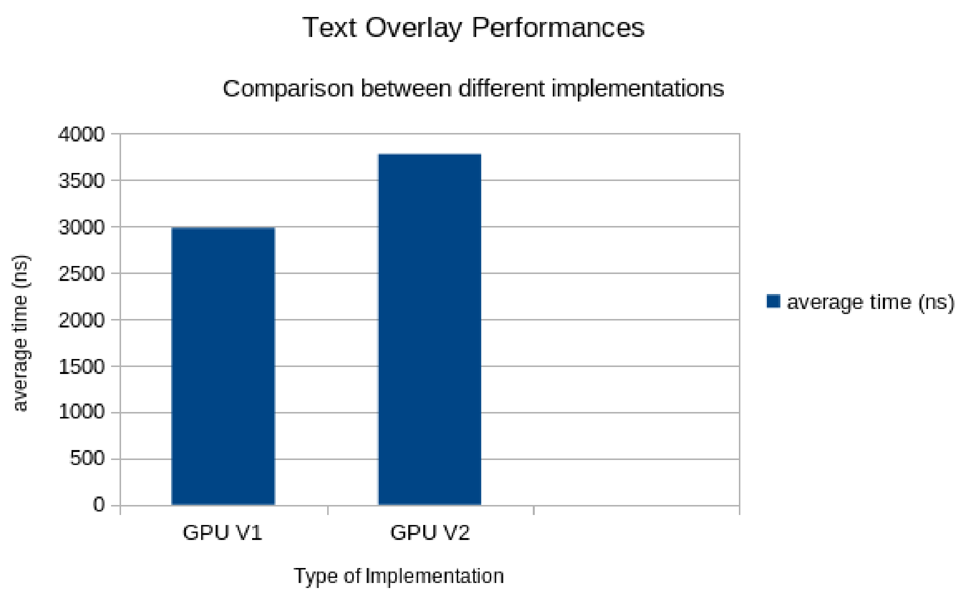

Figure 11: *Text Overlay Comparison*

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:

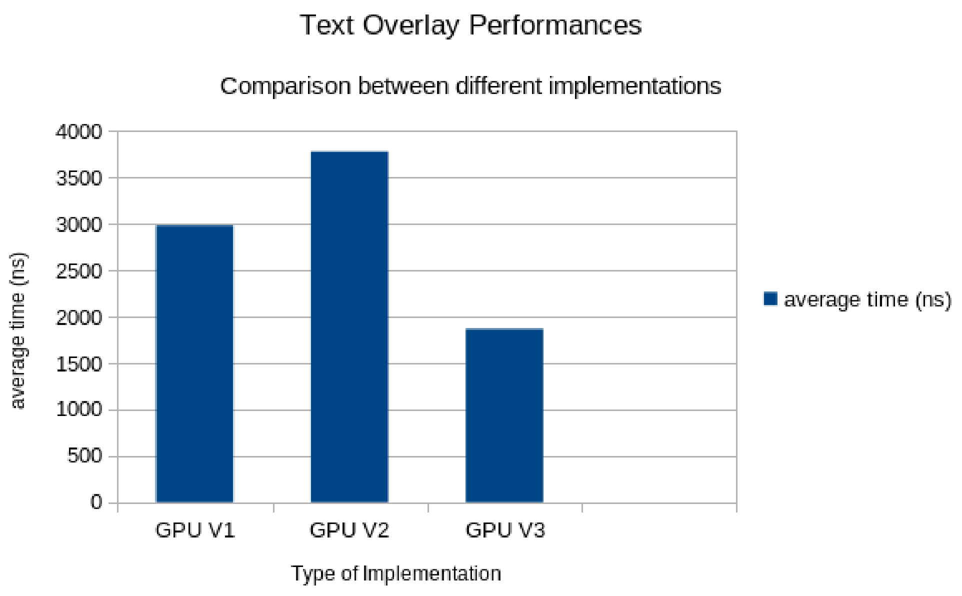

Figure 12: *Text Overlay Comparison*

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:

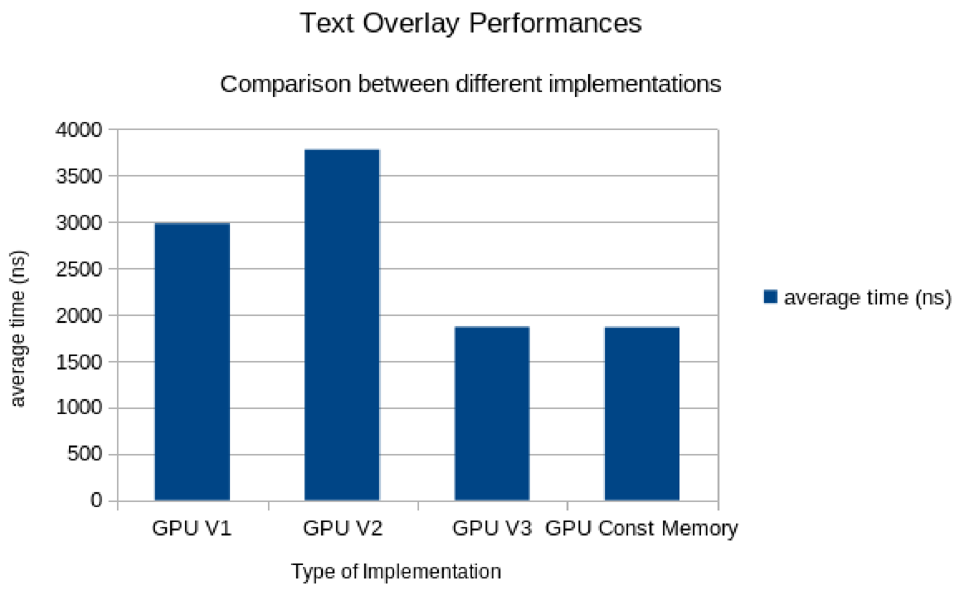

Figure 13: *Text Overlay Comparison*