Politecnico di Torino
GPU Programming

REPORT: PROJECT
WEBCAM STREAMING & FILTERS

December 22, 2021

Matteo Battilana, Salvatore Gabriele La Greca, Giovanni Pollo

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

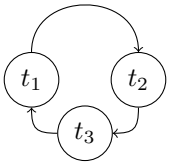
Before looking at the algorithm that computes the difference between frames, is important to give a look at how frames are captured and the overall architecture of the software. It's a client server architecture where the server transmits the difference to the client via a socket.

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` it's a good difference so it can be sent. 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.

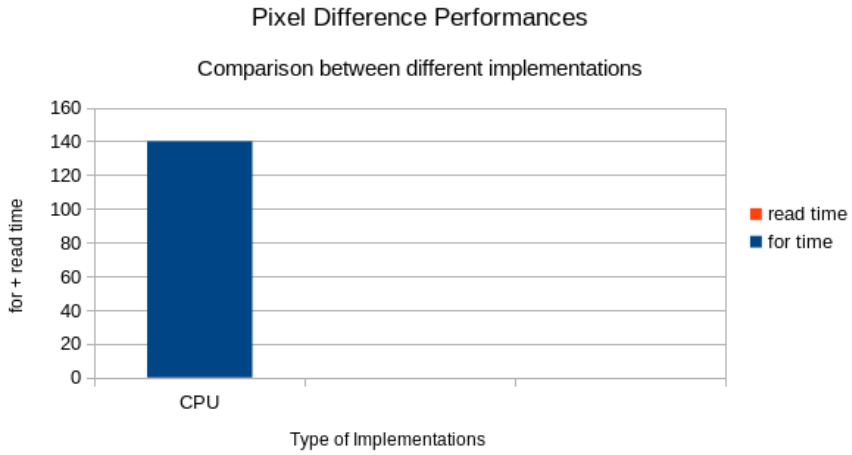


Figure 3: *Pixel Difference Comparison*

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:

```
1  struct cudaDeviceProp prop;
2  cudaGetDeviceProperties(&prop, 0); // retrieves device infos
3
4  int total = 3 * ctx.sampleMat->rows * ctx.sampleMat->cols; // no. bytes
5  int nMaxThreads = prop.maxThreadsPerBlock;
6  int maxAtTime = total / nMaxThreads;
7
```

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:

```
1  uint8_t *d_current, *d_previous, *d_diff;
2  int *d_xs;
3  unsigned int *d_pos;
4
5  cudaMalloc((void **)&d_diff, total * sizeof *d_diff);
6  cudaMalloc((void **)&d_xs, total * sizeof *d_xs);
7  cudaMalloc((void **)&d_current, total * sizeof *d_current);
8  cudaMalloc((void **)&d_previous, total * sizeof *d_previous);
9  cudaMalloc((void **)&d_pos, sizeof *d_pos);
10
```

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 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.

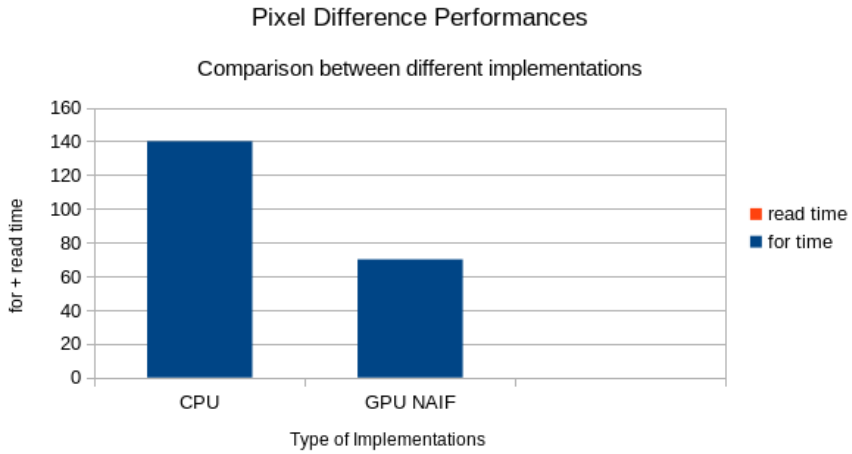


Figure 4: *Pixel Difference Comparison*

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 GPU - async and removal of usefull 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 can be done at the beginning of the kernel and it's easily done even in a concurrent environment by letting doing the reset only at one kernel only. This can be achieved thanks to 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 achieving 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.

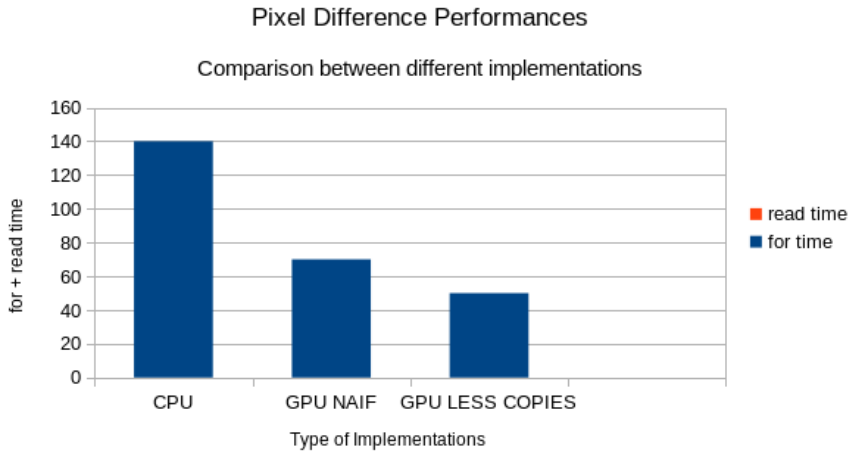


Figure 5: *Pixel Difference Comparison*

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

TODO: show a DMA image, explain why it's better the page locked bla bla

TODO:

- Copying both previous and current everytime and no fixed page host allocation
- Copying only the current and the previous is a swap pointer, still no fixed page host allocation
- Page host allocation
- Copy of all d_diff and d_xs
- Copy first d_pos then a portion of d_diff and d_xs according to d_pos
- Access int-by-int
- AtomicInc in global memory and in shared memory, there are differences?
- Why 512 kernels is best instead of 1024?