Politecnico di Torino
GPU Programming

# ASSIGNMENT: CUDA VIDEO STREAMING

December 24, 2021

Group 16

# 1 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 1 shows a naif implementation of heat map, available at `heat_map_benchmark/v0.cu`, that generates in real time the image on the right.

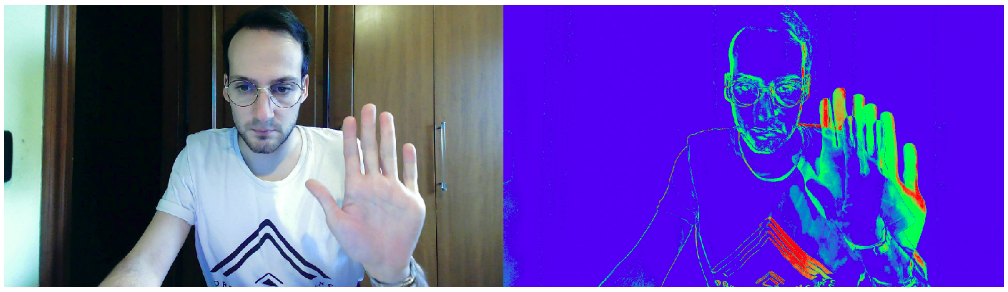


Figure 1: *Example of real-time heat map from webcam*

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.

## 1.1 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 2.

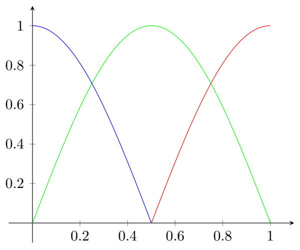


Figure 2: *Mapping function from pixel difference to RGB components*

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.

## 1.2 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 reservation 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 1.3.

### 1.2.1 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, maxSection, 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*2.0);

    // 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 |

Figure 3: *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.

Even a very simple naif CUDA implementation can achieve a large performance improvement with the respect to the CPU. This is thanks to the Single Precision Intrinsics functions of CUDA. More precisely, the sine is computed by using the `__sinf`, that allows to calculate the fast approximate sine of the input argument.

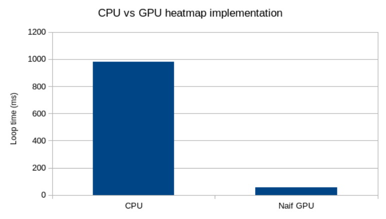


Figure 4: *Loop time CPU vs GPU*

### 1.2.2 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 |

Figure 5: *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).

### 1.2.3 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 |

Figure 6: *Profiling result v3.cu*

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.

## 1.3 Evaluation of the number of threads

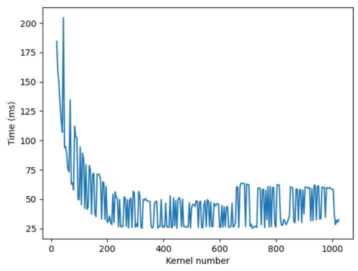
For a first implementation, the number of thread has been set to the maximum allowable from the architecture, that is given by the `cudaGetDeviceProperties` CUDA function in order to make the program independent form the device used. For example, for the Jetson Nano, the maximum number of threads are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

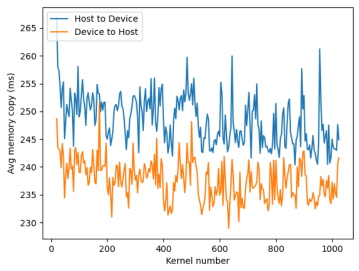
This is the bash code used to extract the *nvprof* information:

```bash
#!/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.

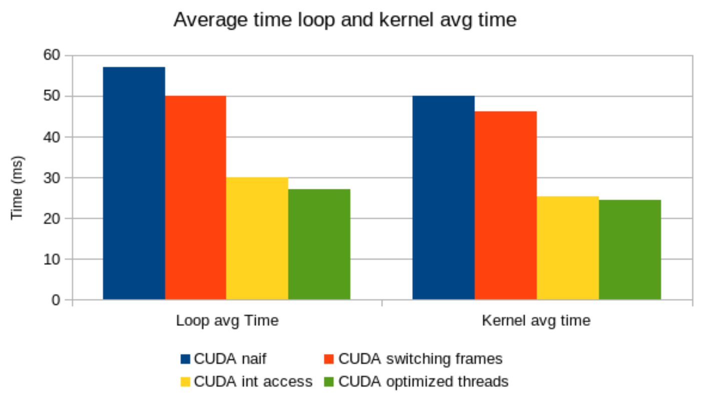
The following table shows some meaningful data extracted from the profiler:

| **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 |

Figure 8: *Profiling result v3.cu*

### 1.3.1 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.



## 2 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.

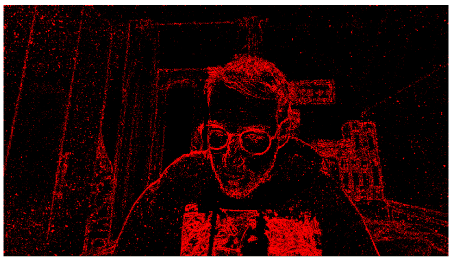


Figure 9: *All non normalized pixel difference are turned to red if above a certain threshold, in this case 20*

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.

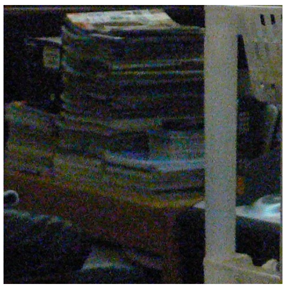
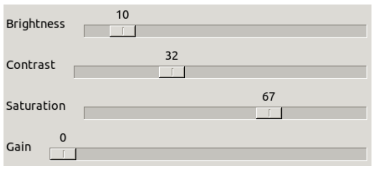


Figure 10: *Example of noise*

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

11, 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.



Brightness 10
Contrast 32
Saturation 67
Gain 0

Figure 11: *Image Control panel*

### 2.0.1 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 9. 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.

### 2.0.2 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;
  for (int i = start; i < max; i++) {
    cc = ((chunk_t *)current)[i];
    pc = ((chunk_t *)previous)[i];
    for (int j = 0; j < sizeof cc; j++) {
      int8_t df = ((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j];

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

      // Access to the global memory
      if((i*(sizeof cc)+j) % 3 == 2){
        d_heat_pixels[i*(sizeof cc)+j] = redColor;
        redColor = 0;
      } else {
        d_heat_pixels[i*(sizeof cc)+j] = 0;
      }
    }
  }
}
```

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*(sizeof cc) + j % 3 == 2` (the red color).

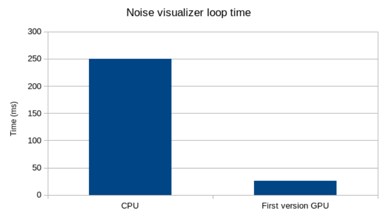
This implementation allows to copy the frame from Host to Device, compute the noise visualization and retrieve back the image from Device to Host memory in around 26ms.

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

| Type | Time (%) | Avg | Min | Max | Name |
|---|---|---|---|---|---|
| GPU activities | 81.38 | 21.759 | 19.489 | 32.94 | kernel_red |
| | 9.59 | 2.5469 | 2.2134 | 6.8552 | [CUDA memcpy HtoD] |
| | 9.03 | 2.4141 | 2.1828 | 3.5408 | [CUDA memcpy DtoH] |
| API calls | 87.41 | 13.992 | 2.7579 | 156.01 | cudaMemcpy |
| | 12.13 | 194.86 | 3.8501 | 576.73 | cudaMalloc |

Figure 12: *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.



### 2.0.3 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:

```
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;
  for (int i = start; i < max; i++) {
    cc = ((chunk_t *)current)[i];
    pc = ((chunk_t *)previous)[i];
    for (int j = 0; j < sizeof cc; j++) {
      int8_t df = ((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j];

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

      // Access to the global memory
      if((i*(sizeof cc)+j) % 3 == 2){
        d_heat_pixels[i*(sizeof cc)+j] = redColor;
        redColor = 0;
      }
    }
  }
}
```

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 | 16.684 | 16.372 | 35.942 | kernel_red |
| | 12.37 | 2.6294 | 2.2265 | 6.3841 | [CUDA memcpy HtoD] |
| | 12.00 | 2.6294 | 2.2265 | 6.3841 | [CUDA memcpy DtoH] |
| API calls | 86.00 | 11.610 | 2.6999 | 41.414 | cudaMemcpy |
| | 13.56 | 183.71 | 3.7352 | 543.50 | cudaMalloc |

Figure 13: *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:

```
cuobjdump ./a.out -xelf all
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 14: 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 15: Assembly extract of the second CUDA implementation of the noise visualizer

### 2.0.4 Fastest 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 16.

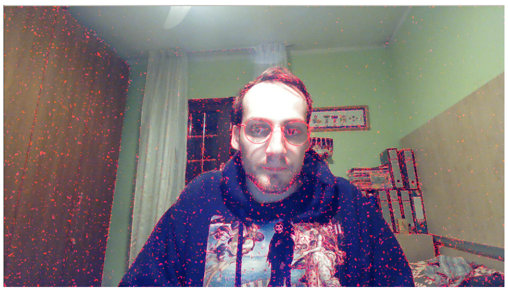


Figure 16: *Red pixels over the frame*

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:

```
typedef int4 chunk_t;

__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++) {

    cc = ((chunk_t *)current)[i];
    pc = ((chunk_t *)previous)[i];
    for (int j = 0; j < sizeof cc; j++) {
      int8_t df = ((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j];

      if (df < -LR_THRESHOLDS || df > LR_THRESHOLDS) {
        toUpdate = true;
      }

      // Update only if needed
      if (toUpdate && (i*(sizeof cc)+j) % 3 == 2){
      previous[i*(sizeof cc)+j] = 255;
      toUpdate = false;
    }
  }
}
```

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 | 7.5647 | 4.9421 | 51.426 | kernel_red |
| | 23.69 | 3.3450 | 2.4960 | 6.6161 | [CUDA memcpy HtoD] |
| | 23.26 | 3.3160 | 2.8359 | 4.2297 | [CUDA memcpy DtoH] |
| API calls | 81.86 | 7.7539 | 2.9697 | 57.082 | cudaMemcpy |
| | 17.51 | 166.67 | 5.4298 | 327.91 | cudaMalloc |

Figure 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.

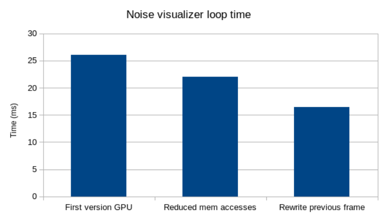


Figure 18: *Comparison between GPU versions for noise visualizer*