
**ASSIGNMENT:
CUDA VIDEO STREAMING**

December 23, 2021

1 Introduction

2 HeatMap

Since the basis of this project is to send the pixel difference, the lower the dissimilarities the higher the bandwidth saving. So, 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 represents the magnitude of a measurement as color in two dimensions, in this case an image. The idea is first compute the difference 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 naive implementation of heat map, available at [heat_map_benchmark/v0.cu](https://github.com/heat-map-benchmark/v0.cu), that generates in real time the image on the right.

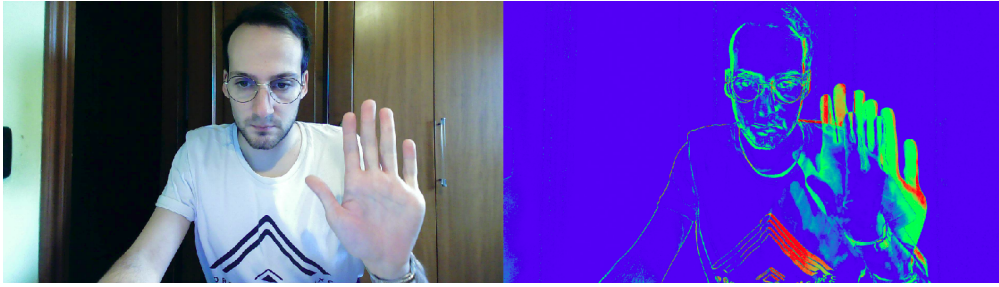


Figure 1: *Example of real-time heat map from webcam*

The first *naif* version is running on the CPU and 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.

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

2.1 Heat map pixel mapping - CPU Naif implementation

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 the $Previous[i, R]$ 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 three 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 function, 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.

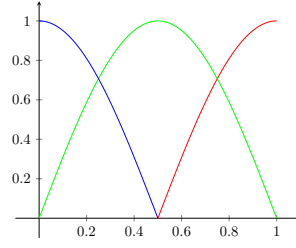


Figure 2: Mapping function from pixel difference to RGB components

The CPU implementation is the following:

```

1 struct HeatElemt {
2     int r;
3     int g;
4     int b;
5 };
6
7 HeatElemt getHeatPixel(int diff){
8     struct HeatElemt h;
9     float diff1 = diff/(255.0*2.0);
10
11     // Map the difference into the three color components
12     h.r = min(max(sin(M_PI*diff1 - M_PI/2.0)*255.0, 0.0),255.0);
13     h.g = min(max(sin(M_PI*diff1)*255.0, 0.0),255.0);
14     h.b = min(max(sin(M_PI*diff1 + M_PI/2.0)*255.0, 0.0),255.0);
15 }
16
17 while (1) {
18     cap >> image2;
19     for (int y = 0; y < H; y++){
20         for (int x = 0; x < W; x++){
21             Vec3b & intensity = image1.at<Vec3b>(y, x);
22             Vec3b a = image1.at<Vec3b>(y, x);
23             Vec3b b = image2.at<Vec3b>(y, x);
24
25             // Compute the absolute difference
26             HeatElemt elem = getHeatPixel(abs(a.val[0] - b.val[0]) +
27             abs(a.val[1] - b.val[1]) + abs(a.val[2] - b.val[2]));
28
29             intensity.val[0] = elem.b;
30             intensity.val[1] = elem.g;
31             intensity.val[2] = elem.r;
32         }
33     }
34     image1 = image2.clone();
35 }

```

2.2 CUDA heat map 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.

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 launched, 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.

2.2.1 CUDA first naif version

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.

Unfortunately, the 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 one and using a shared memory would have only reduced the performance due to an useless copy and there are not portion of constant memory.

CUDA allows to get the information about the maximum number of thread by using the `cudaGetDeviceProperties` command. Having the maximum number of thread, it's possible to call the kernel in the following way:

```

1 // Kernel per block computation
2 cudaGetDeviceProperties(&prop, 0);
3 int threads = prop.maxThreadsPerBlock;
4
5 // Copy from Host to Device
6 cudaMemcpy(d_prev, image1, W*H*C * sizeof *image1, cudaMemcpyHostToDevice);
7 cudaMemcpy(d_curr, image2, W*H*C * sizeof *image2, cudaMemcpyHostToDevice);
8
9 kernel<<<1, threads>>>(d_curr, d_prev, (W*H*C)/threads, d_out);
10
11 // Copy heat map from Device to Host
12 cudaMemcpy(heatmap, d_out, W*H*C * sizeof *heatmap, cudaMemcpyDeviceToHost);

```

An extensive analysis of the correct number of threads has been done at Section 2.3.

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.

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

```

1 __global__ void kernel(uint8_t *current, uint8_t *previous,
2                       int maxSect, uint8_t* d_heat_pixels) {
3
4     // Index relative to the block
5     int x = threadIdx.x + blockDim.x * blockIdx.x;
6
7     // Start of the sector for this thread
8     int start = x * maxSect;
9     int max = start + maxSect;
10    for (int i = start; i < max; i=i+C) {
11
12        // Compute the pixel difference
13        int pixelDiff = fabsf(current[i] - previous[i]) + fabsf(current[i+1]
14        - previous[i+1]) + fabsf(current[i+2] - previous[i+2]);
15        float diff1 = pixelDiff/(255*2.0);
16
17        // Map different into the three color component
18        int r = fminf(fmaxf(sinf(M_PI*diff1 - M_PI/2.0)*255.0, 0.0), 255.0);
19        int g = fminf(fmaxf(sinf(M_PI*diff1)*255.0, 0.0), 255.0);
20        int b = fminf(fmaxf(sinf(M_PI*diff1 + M_PI/2.0)*255.0, 0.0), 255.0);
21        d_heat_pixels[i] = b;
22        d_heat_pixels[i+1] = g;
23        d_heat_pixels[i+2] = r;
24    }
25 }

```

After having run the `nvprof`, the main contribution 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.

2.2.2 CUDA switching frames

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.

```

1 while(1){
2     // New frame fetch
3     cap >> image2;
4
5     // Pointer switching
6     uint8_t* tmp = d_curr;
7     d_curr = d_prev;
8     d_prev = tmp;
9
10    // Kernel call
11    cudaMemcpy(d_curr, image2, W*H*C * sizeof *image2, cudaMemcpyHostToDevice);
12    kernel<<<1, threads>>>(d_curr, d_prev, (W*H*C)/threads, d_out);
13    cudaMemcpy(heatmap, d_out, W*H*C * sizeof *heatmap, cudaMemcpyDeviceToHost);
14
15    image1 = image2.clone();
16 }

```

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 4: *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).

2.2.3 CUDA access at the int level instead of uint8_t

In order to reduce the number of accesses to 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 only the first three bytes (RGB), in order to optimize and avoid to read twice from the memory, colors are read 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.

```

1 __global__ void kernel(uint8_t *current, uint8_t *previous,
2     int maxSect, uint8_t* d_heat_pixels) {
3     int x = threadIdx.x + blockDim.x * blockIdx.x;
4     int start = x * maxSect;
5     int max = start + maxSect;

```

```

6   int cc, pc;
7   for (int i = start; i < max; i++) {
8
9       // Access one 4 byte at a time
10      cc = ((int *)current)[i];
11      pc = ((int *)previous)[i];
12      int pixelDiff = 0;
13      for (int j = 0; j < 4; j++) {
14
15          // Conversion from difference to heat map only every 3 bytes
16          if((i*4+j) % 3 == 0){
17              int pixelDiff = fabsf(((uint8_t *)&cc)[j] - ((uint8_t *)&pc)[j]) +
18                  fabsf(((uint8_t *)&cc)[j+1] - ((uint8_t *)&pc)[j+1]) +
19                  fabsf(((uint8_t *)&cc)[j+2] - ((uint8_t *)&pc)[j+2]);
20              float diff1 = pixelDiff/(255*2.0);
21              int r = fminf(fmaxf(sinf(M_PI*diff1 - M_PI/2.0)*255.0, 0.0),255.0);
22              int g = fminf(fmaxf(sinf(M_PI*diff1)*255.0, 0.0),255.0);
23              int b = fminf(fmaxf(sinf(M_PI*diff1 + M_PI/2.0)*255.0, 0.0),255.0);
24              d_heat_pixels[i*4+j] = b;
25              d_heat_pixels[i*4+j+1] = g;
26              d_heat_pixels[i*4+j+2] = r;
27
28              // Reset the pixel difference
29              pixelDiff = 0;
30          }
31      }
32  }
33  }

```

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

```

1   cudaGetDeviceProperties(&prop, 0);
2   int threads = prop.maxThreadsPerBlock;
3
4   cudaMemcpy(d_curr, image2, W*H*C * sizeof *image2, cudaMemcpyHostToDevice);
5
6   // Kernel call /4
7   kernel<<<1, threads>>>(d_curr, d_prev, ((W*H*C)/threads)/4, d_out);
8   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 5: *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.

2.3 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.

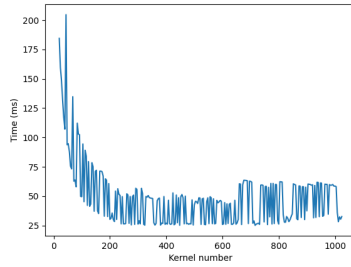
This is the bash code to execute the

```

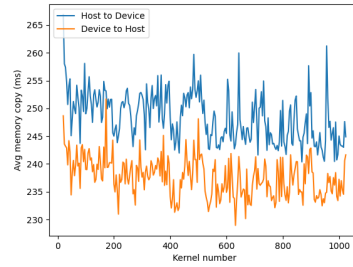
1  #!/bin/bash
2  for i in {1..256}
3  do
4      k=$(( 4*i ))
5
6      # Profiler call
7      avg='sudo /usr/local/cuda/bin/nvprof ./heatMap $k 2>&1'
8
9      # Data extraction
10     kern='echo "$avg" | grep kernel | awk '{print $6}''
11     kernt='echo "$avg" | grep kernel | awk '{print $4}''
12     hd1='echo "$avg" | grep "CUDA memcpy HtoD" | awk '{print $4}''
13     hd2='echo "$avg" | grep "CUDA memcpy HtoD" | awk '{print $2}''
14     dh1='echo "$avg" | grep "CUDA memcpy DtoH" | awk '{print $4}''
15     dh2='echo "$avg" | grep "CUDA memcpy DtoH" | awk '{print $2}''
16     echo "$k $all $kern $kernt $hd1 $hd2 $dh1 $dh2" >> times.txt
17 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.



(a) Time needed to perform an heatmap depend-
ing 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 thread 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.

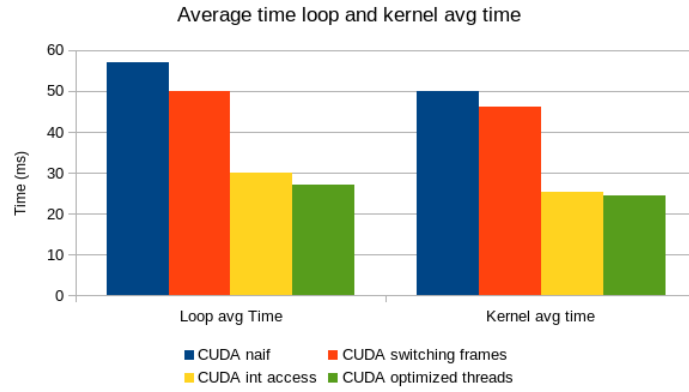
In fact, by running the same exact algorithm describe before but the number of threads is set to 718 (that is the best kernel accordingly to the plot above), the time needed to copy a frame, compute the heat map and then copy back the heat map matrix is about 27ms. This leads to a increasing of performance 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 7: *Profiling result v3.cu*

By considering only the CUDA algorithm, the optimizations show decreasing fashion of the 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.



3 Noise filtering

A second implementation, that is used to speed up the computation is to turn in red all pixels that are above a certain threshold. Obviously, the complexity of this computation is much easier than the previous one and in the CPU allows to compute that kind of map in around 300ms.



Figure 8: *All non normalized pixel difference are turned to red if above a certain threshold, in this case 30*

The two implementations are managed via some compiler directives that allows to increase the performance, since only one of them can be visualized. The purpose of them is completely different, since the original heat map, the one explained at the beginning 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. This allows to have a visual evaluation of the noise filter, that will be explained in the next section.