Università degli studi di Modena e Reggio Emilia
Dipartimento di Ingegneria “Enzo Ferrari”
Corso di Laurea Magistrale in Ingegneria Informatica

Bird's Eye View Transformation

PROGETTO DI CORSO

Fabrizio Di Blasi | High Performance Computing | a.a 2018/2019

Sommario

Introduzione

a. Finalità del progetto

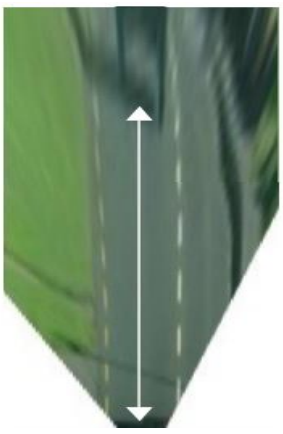
In questa relazione viene descritto il procedimento per effettuare la trasformazione “Bird's Eye View”. Tale trasformazione consiste nel ruotare, secondo dei parametri impostati dall’utente, ogni singolo fotogramma di un video già preesistente, oppure catturato dalla webcam (o videocamera).
In questo modo si ha la possibilità di interpolare e proiettare il video catturato secondo un’angolatura differente da quella realmente ripresa.
L’utente ha la possibilità di effettuare la trasposizione dei pixel tramite:

- Funzioni già esistenti di OpenCV
- Funzioni accelerate in CUDA

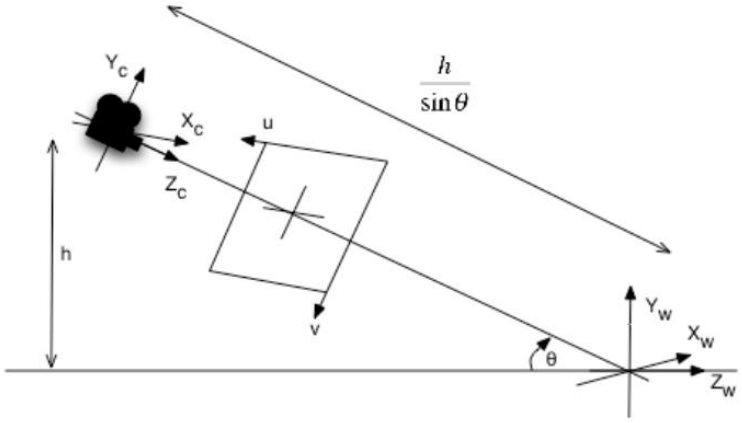
Tale trasposizione deve essere effettuata nel più beve tempo possibile in modo da non creare troppo ritardo, ed un calo degli FPS. Dopo una introduzione pratica del lavoro, verranno esposti i risultati sperimentali.

b. Stato dell’arte

La “Bird's Eye View” fa parte della Inverse Perspective Mapping (IPM) che consiste in una tecnica matematica in cui le coordinate di un sistema vengono trasformate da una prospettiva all’altra. La IPM può essere usata in ambito automotive per ottenere una visione top-down, ad esempio :



Dalle due figure si può notare lo scopo principale, ovverosia, la proiezione di un fotogramma di una dashcam secondo una visuale top-down. Per creare tale visualizzazione, si ha il bisogno di trovare il corretto mapping di un punto della superficie (x,y,z) con il piano (u,v), noto l’angolo θ.



La riproiezione può essere calcolata tramite la seguente formula:

$(u,v,1)^T = K\ T\ R(x,y,z,1)^T$

dove:

R : matrice di rotazione

T : matrice di traslazione

K : matrice contenente i parametri della fotocamera

$$R = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sen\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{matrix} \qquad T = \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -h/sen\theta \\ 0 & 0 & 0 & 1 \end{matrix} \qquad K = \begin{matrix} f*ku & s & u_0 & 0 \\ 0 & f*kv & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{matrix}$$

Tramite le seguenti matrici, è possibile calcolare il mapping top down di ciascun singolo pixel, infatti, data *f*, distanza focale della fotocamera, ed *s,* parametro di distorsione della camera, l'equazione :

$$(u,v,1)^T = K\ T\ R\ (\ x,\ y,\ z,\ 1)^T$$

l'equazione può essere riscritta come :

$$\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} * \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

essendo interessati al piano della strada ($Y_w = 0$), la precedente formula diventa:

$$\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix} = \begin{bmatrix} p_{11} & p_{13} & p_{14} \\ p_{21} & p_{23} & p_{24} \\ p_{31} & p_{33} & p_{34} \end{bmatrix} * \begin{bmatrix} X_w \\ Z_w \\ 1 \end{bmatrix}$$

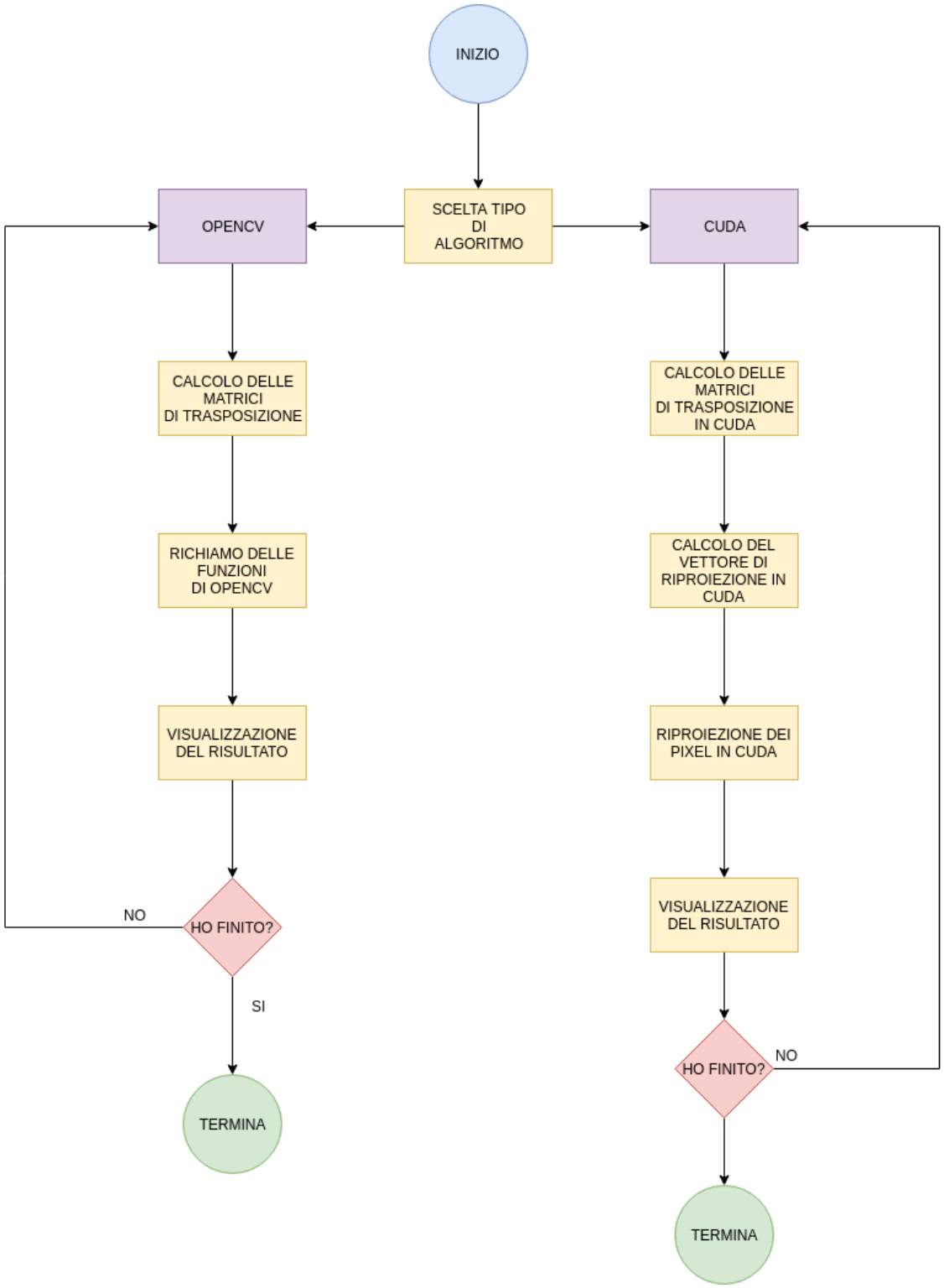
Grazie a quest'ultima equazione è possibile ottenere il mapping top down desiderato.

Visto il numero elevato di prodotti matriciali, si è scelto di parallelizzare il più possibile il calcolo matriciale tramite l'uso di kernel CUDA.

Inoltre, questo procedimento di traslazione deve essere effettuato su ogni singolo canale dell'immagine RGB

Descrizione

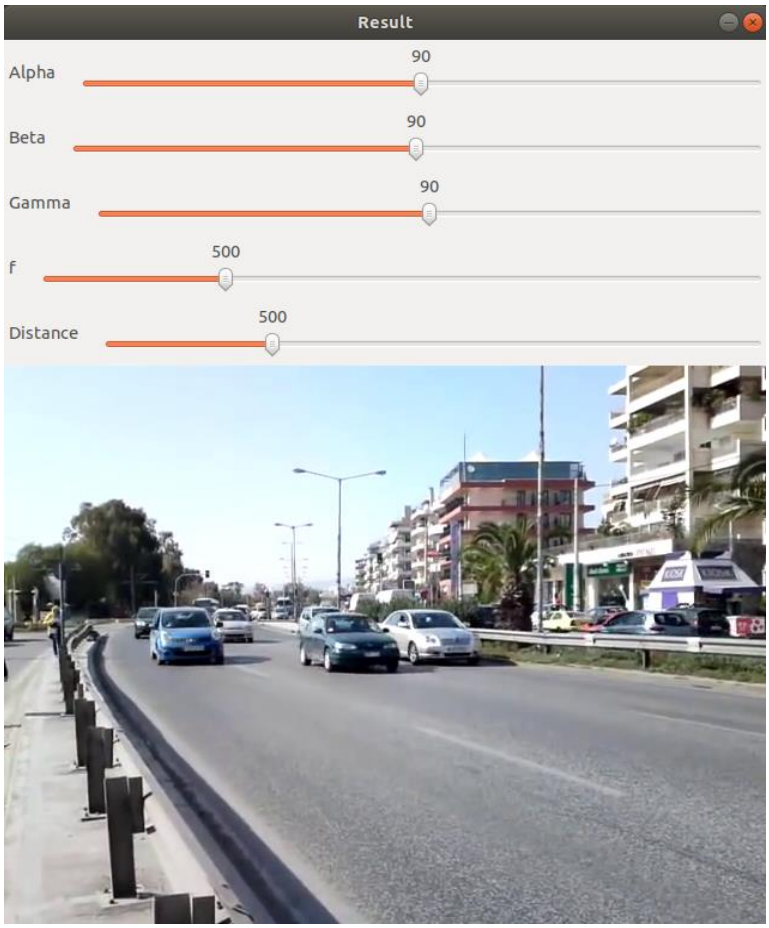
a. Schema a Blocchi



Dallo schema a blocchi presentato è possibile capire le due differenti modalità di esecuzione, selezionabili da linea di comando, del programma e le varie fasi svolte da esso.

Il funzionamento logico dei due rami è pressoché lo stesso, ma, usando il procedimento in CUDA, si ha la problematica di calcolare manualmente le posizioni dei nuovi pixel, per poi effettuare la trasposizione su tutti i 3 canali RGB. Per la memorizzazione del fotogramma sul device sono stati utilizzati degli oggetti definiti da OpenCV, che vanno direttamente ad allocare della memoria nella GPU, manipolabile tramite un kernel CUDA.

b. Guida per L'utente



Interfaccia:

La schermata iniziale è la seguente, l'utente ha a disposizione 5 regolazioni, le prime 3 sono per definire l'angolo con il quale ruotare il fotogramma, mentre gli ultimi due servono ad impostare i parametri della fotocamera.

Tale interfaccia è identica per entrambi i "rami" dell'algoritmo

il programma viene eseguito da linea di comando e si hanno 4 opzioni differenti, in base se si vuole l'algoritmo con solamente chiamate di OpenCV oppure se accelerato in cuda, e se si vuole analizzare lo stream della webcam oppure un video già preesistente.

Se si vuole usare l'accelerazione in CUDA :

$ ./app y

$ ./app y <video path>

Se si vogliono usare solamente direttive di OpenCV:

$ ./app n

$ ./app n <video path>

c. Codice Sorgente e Funzioni Accelerabili

Main:

Il main è la funzione principale, che richiama tutte le altre che analizzano lo stream video. Nella prima parte di codice si può osservare la gestione dell'input da linea di comando e la prevenzione di eventuali errori di richiamo.

All'interno del main viene tenuta traccia del tempo di esecuzione delle due diverse tipologie di funzioni di analisi.

```
1.  int main(int argc, char const *argv[]) {
2.
3.      if(argc > 3 || argc == 1) {
4.        cerr << "Usage: " << argv[0] << " < CUDA : y / n > <' /path/to/video/ ' | nothing >
    " << endl;
5.        cout << "Exiting...." << endl;
6.        return -1;
7.      }
8.      int flag=0;
9.      Mat image,output;
10.
11.     VideoCapture capture;
12.     string cudaflag = argv[1];
13.     if (cudaflag == "y"){
14.         CUDA = true;
15.         cout<<"** CUDA ON ** \n";
16.     }else{
17.         CUDA = false;
18.         cout<<"** CUDA OFF ** \n";
19.
20.     }
21.
22.     if (argc == 2){
23.         capture.open(0);
24.     }
25.     if (argc == 3){
26.         string filename = argv[2];
27.         capture.open(filename);
28.     }
29.
30.     if(!capture.isOpened()) throw "Error reading video";
31.
32.     namedWindow("Result", 1);
33.
34.     createTrackbar("Alpha", "Result", &alpha_, 180);
35.     createTrackbar("Beta", "Result", &beta_, 180);
36.     createTrackbar("Gamma", "Result", &gamma_, 180);
37.     createTrackbar("f", "Result", &f_, 2000);
38.     createTrackbar("Distance", "Result", &dist_, 2000);
39.
40.     cout << "Capture is opened" << endl;
41.     for(;;)
42.     {
43.         capture >> image;
44.         //stampo il tipo di immagine
45.         if(flag == 0){
46.             string ty = "CV_" + type2str( image.type() );
47.             cout << "tipo matrice :" << ty.c_str() <<endl;
```

```
48.             flag = 1;
49.         }
50.         resize(image, image,Size(FRAMEWIDTH, FRAMEHEIGHT));
51.
52.         if (CUDA){
53.             std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now()
    ;
54.             CUDA_birdsEyeView(image, output);
55.             std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();

56.             std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::m
    icroseconds>(end - begin).count() << "[µs]" << std::endl;
57.             std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::n
    anoseconds> (end - begin).count() << "[ns]" << std::endl;
58.         }else{
59.             std::chrono::steady_clock::time_point begin = std::chrono::steady_clock::now()
    ;
60.             birdsEyeView(image, output);
61.             std::chrono::steady_clock::time_point end = std::chrono::steady_clock::now();

62.             std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::m
    icroseconds>(end - begin).count() << "[µs]" << std::endl;
63.             std::cout << "Time difference = " << std::chrono::duration_cast<std::chrono::n
    anoseconds> (end - begin).count() << "[ns]" << std::endl;
64.         }
65.         //per la visualizzazione
66.         if(output.empty())
67.             break;
68.         //drawText(image);
69.         imshow("Result", output);
70.         if(waitKey(10) >= 0)
71.             break;
72.     }
73.
74.
75.     return 0;
76. }
```

Funzioni accelerabili e non:

Lo scopo principale del progetto è stato quello di parallelizzare il più possibile il codice sequenziale, per farlo si è intervenuti sui vari cicli presenti all'interno del codice.

La seguente funzione è la versione non parallelizzata che non sfrutta CUDA, ma bensì solo operazioni definite dalla libreria OpenCV. Infatti, per il calcolo dei prodotti matriciali e per il *warping* dell'immagine non è stata effettuato alcun calcolo parallelo, i prodotti matriciali sono svolti tramite l'overload dell'operatore " * ", la traslazione invece tramite la funzione "*warpPerspective()*"

```
1.  void birdsEyeView(const Mat &input, Mat &output){
2.      double focalLength, dist, alpha, beta, gamma;
3.
4.      alpha =((double)alpha_ -90) * PI/180;
5.      beta =((double)beta_ -90) * PI/180;
6.      gamma =((double)gamma_ -90) * PI/180;
7.      focalLength = (double)f_;
8.      dist = (double)dist_;
9.
10.     Size input_size = input.size();
```

```
    double w = (double)input_size.width, h = (double)input_size.height;


    // Projecion matrix 2D -> 3D
    Mat A1 = (Mat_<float>(4, 3)<<
        1, 0, -w/2,
        0, 1, -h/2,
        0, 0, 0,
        0, 0, 1 );

    // Rotation matrices Rx, Ry, Rz
    Mat RX = (Mat_<float>(4, 4) <<
        1, 0, 0, 0,
        0, cos(alpha), -sin(alpha), 0,
        0, sin(alpha), cos(alpha), 0,
        0, 0, 0, 1 );

    Mat RY = (Mat_<float>(4, 4) <<
        cos(beta), 0, -sin(beta), 0,
        0, 1, 0, 0,
        sin(beta), 0, cos(beta), 0,
        0, 0, 0, 1  );

    Mat RZ = (Mat_<float>(4, 4) <<
        cos(gamma), -sin(gamma), 0, 0,
        sin(gamma), cos(gamma), 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1  );
    // R - rotation matrix
    Mat R = RX * RY * RZ;
    // T - translation matrix
    Mat T = (Mat_<float>(4, 4) <<
        1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, dist,
        0, 0, 0, 1);
    // K - intrinsic matrix
    Mat K = (Mat_<float>(3, 4) <<
        focalLength, 0, w/2, 0,
        0, focalLength, h/2, 0,
        0, 0, 1, 0
        );
    Mat transformationMat = K * (T * (R * A1));
    warpPerspective(input, output, transformationMat, input_size, INTER_CUBIC | WARP_INVER
SE_MAP);
    return;
}
```

<u>Versione accelerata:</u>

Di seguito viene presentato il punto chiave del progetto, ovvero la parallelizzazione del codice precedente tramite l'uso di CUDA.

Tale tipologia di parallelizzazione obbliga il programmatore ad una gestione esplicita della memoria:

Come si può vedere dall'immagine precedente, le aree di memoria della GPU e della CPU sono distaccate. Per accedere alla GPU ci si deve affidare al bus di comunicazione che introduce sia una latenza nel trasferimento, e sia una limitazione in termini di banda.

Perciò è utile utilizzare un *offloading programming model* solamente quando la quantità di dati da analizzare è elevato in modo da mitigare le tempistiche necessarie per lo scambio di informazioni tra *device (GPU)* ed *host (CPU)*.

CUDA_birdsEyeView():

Questa funzione è la versione parallelizzata della precedente, come si può vedere, non compaiono più oggetti di tipo `cv::Mat` per il calcolo delle matrici di traslazione, ma solamente dei vettori, che vengono poi affidati alla funzione `matrixMultiplication()`. Tale funzione è un *wrapper* del *kernel CUDA*, che viene esposto subito dopo.

```
1.  void CUDA_birdsEyeView(const Mat &input, Mat &output){
2.
3.      cudaError_t error;
4.
5.      double focalLength, dist, alpha, beta, gamma;
6.
7.      alpha =((double)alpha_ -90) * PI/180;
8.      beta =((double)beta_ -90) * PI/180;
9.      gamma =((double)gamma_ -90) * PI/180;
10.     focalLength = (double)f_;
11.     dist = (double)dist_;
12.
13.     Size input_size = input.size();
14.     double w = (double)input_size.width, h = (double)input_size.height;
15.
16.
17.     float A1[12] = {
18.         1, 0, -w/2,
19.         0, 1, -h/2,
20.         0, 0, 0,
21.         0, 0, 1
22.     };
23.
24.     float RX[16] = {
25.         1, 0, 0, 0,
26.         0, cos(alpha), -sin(alpha), 0,
27.         0, sin(alpha), cos(alpha), 0,
28.         0, 0, 0, 1
29.     };
```

```
30.
31.     float RY[16] ={
32.         cos(beta), 0, -sin(beta), 0,
33.         0, 1, 0, 0,
34.         sin(beta), 0, cos(beta), 0,
35.         0, 0, 0, 1
36.     };
37.
38.     float RZ[16] = {
39.         cos(gamma), -sin(gamma), 0, 0,
40.         sin(gamma), cos(gamma), 0, 0,
41.         0, 0, 1, 0,
42.         0, 0, 0, 1
43.     };
44.
45.     // cout << "stampo RX \n";
46.     // stampaMatrice(RX , 4, 4);
47.     // R - rotation matrix
48.     // Mat R = RX * RY * RZ;
49.
50.     float R[16], XY[16];
51.     error = matrixMultiplication(RX, RY, XY, 4, 4, 4, 4);
52.     if (error != cudaSuccess) {
53.         fprintf(stderr, "cudaMalloc failed!");
54.         exit(0);
55.     }
56.     error = matrixMultiplication(XY, RZ, R, 4, 4, 4, 4);
57.     if (error != cudaSuccess) {
58.         fprintf(stderr, "cudaMalloc failed!");
59.         exit(0);
60.     }
61.
62.     // T - translation matrix
63.     float T[16] = {
64.         1, 0, 0, 0,
65.         0, 1, 0, 0,
66.         0, 0, 1, dist,
67.         0, 0, 0, 1
68.     };
69.     // K - intrinsic matrix
70.     float K[12] = {
71.         focalLength, 0, w/2, 0,
72.         0, focalLength, h/2, 0,
73.         0, 0, 1, 0
74.     };
75.
76.     //Mat transformationMat = K * (T * (R * A1));
77.     float R_A1[12], T_RA1[12], transformationvector[9];
78.
79.     error = matrixMultiplication(R, A1, R_A1, 4, 4, 4, 3);
80.     if (error != cudaSuccess) {
81.         fprintf(stderr, "cudaMalloc failed!");
82.         exit(0);
83.     }
84.     // cout << "R * A1 \n";
85.     // stampaMatrice(R_A1, 4, 3);
86.
87.     error = matrixMultiplication(T, R_A1, T_RA1, 4, 4, 4, 3);
88.     if (error != cudaSuccess) {
89.         fprintf(stderr, "cudaMalloc failed!");
90.         exit(0);
91.     }
92.     // cout << "T* (R * A1) \n";
93.     // stampaMatrice(T_RA1, 4, 3);
94.
95.     error = matrixMultiplication(K, T_RA1, transformationvector, 4, 4, 4, 3);
96.     if (error != cudaSuccess) {
```

```
97.         fprintf(stderr, "cudaMalloc failed!");
98.         exit(0);
99.     }
100.
101.     cv::Mat tranf_mat(3,3,CV_32FC1);
102.
103.     arrayToMat(tranf_mat,transformationvector,9);
104.
105.     warpPerspectiveRemappingCUDA(input, output, tranf_mat);
106.
107.     return;
108.
109. }
```

matrixMultiplication:

Nella seguente funzione è possibile chiarire meglio il concetto dell'*offloading programming model* e di come si utilizza *CUDA*.

A riga 1, si può vedere il *kernell CUDA*, esso ha il compito di effettuare i prodotti matriciali, sfruttando gli indici di riga e colonna di uno specifico thread del blocco.

La potenzialità di *CUDA* risiede proprio in questa funzionalità, ovverosia, assegnare il lavoro in base all'indice del thread. Ma come esposto prima, si ha il bisogno di gestire esplicitamente la memoria della GPU, ciò viene effettuato dal *wrapper* che tramite le `cudaMalloc` e `cudaMemcpy` effettuano tali operazioni. La prima assegna una specifica area di memoria sul *device* la seconda, invece, sposta i dati dall'*host* al *device*. Tale spostamento dovrà essere fatto anche in direzione opposta una volta terminata la computazione.

```
1.  __global__ void generic_mat_mul(float *A, float *B, float *C, int numARows,int numAColumns
    , int numBRows, int numBColumns) {
2.     int row = blockIdx.y * blockDim.y + threadIdx.y;
3.     int col = blockIdx.x * blockDim.x + threadIdx.x;
4.     if (row < numARows && col < numBColumns) {
5.         float sum = 0;
6.         for (int ii = 0; ii < numAColumns; ii++) {
7.             sum += A[row * numAColumns + ii] * B[ii * numBColumns + col];
8.         }
9.         C[row * numBColumns + col] = sum;
10.    }
11. }
12.
13. cudaError_t matrixMultiplication(float *A, float *B, float *C, int numARows,int numAColumn
    s, int numBRows, int numBColumns){
14.
15.    dim3 blockDim(16, 16);
16.    dim3 gridDim(ceil(((float)numAColumns) / blockDim.x),ceil(((float)numBRows) / blockDim
    .y));
17.
18.    float *d_A, *d_B, *d_C;
19.    cudaStatus = cudaMalloc((void **) &d_A, sizeof(float)*numARows*numAColumns);
20.    if (cudaStatus != cudaSuccess) {
21.        fprintf(stderr, "cudaMalloc failed!");
22.        goto Error;
23.    }
24.    cudaStatus = cudaMalloc((void **) &d_B, sizeof(float)*numBRows*numBColumns);
25.    if (cudaStatus != cudaSuccess) {
26.        fprintf(stderr, "cudaMalloc failed!");
27.        goto Error;
28.    }
29.    cudaStatus = cudaMalloc((void **) &d_C, sizeof(float)*numARows * numBColumns);
30.    if (cudaStatus != cudaSuccess) {
31.        fprintf(stderr, "cudaMalloc failed!");
```

```
        goto Error;
    }
    //copio i vettori
    cudaMemcpy(d_A,A,sizeof(float)*numARows*numAColumns,cudaMemcpyHostToDevice);
    cudaMemcpy(d_B,B,sizeof(float)*numBRows*numBColumns,cudaMemcpyHostToDevice);
    cudaMemset(d_C, 0, numARows * numBColumns * sizeof(float));

    generic_mat_mul<<<gridDim, blockDim>>>(d_A, d_B, d_C, numARows, numAColumns, numBRows, numBColumns);
    cudaThreadSynchronize();
    cudaStatus = cudaGetLastError();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "Kernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
        goto Error;
    }

    cudaStatus = cudaDeviceSynchronize();
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);
        goto Error;
    }

    cudaStatus = cudaMemcpy(C, d_C,numARows * numBColumns * sizeof(float), cudaMemcpyDeviceToHost);
    if (cudaStatus != cudaSuccess) {
        fprintf(stderr, "cudaMemcpy failed!");
        goto Error;
    }
    //@@ Free the GPU memory here
Error:
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
    return cudaStatus;

}
```

warpPerspectiveRemappingCUDA:

Questa funzione è il nocciolo del progetto ed ha il compito di calcolare la traslazione del frame video. Tramite "*calculateTransferArray()*" si vanno a calcolare le nuove posizioni dei pixel sulla base della matrice *H*, che è il risultato dei precedenti prodotti matriciali (K * T * R)

```
cudaError_t warpPerspectiveRemappingCUDA(Mat input, Mat &output, const Mat H){
    cudaError_t cudaStatus;

    // allocate array of all locations
    int Numrows = input.rows;
    int Numcols = input.cols;
    int channels   = input.channels();
    int size = Numrows*Numcols;

    int *TransArry = (int *)malloc(sizeof(int)*size);

    cudaStatus = calculateTransferArray(H,TransArry,Numrows, Numcols);

    cout <<" \n richiamo la funzione per il remapping \n";
    cout <<" \n NUMERO DI CANALI : " << input.channels() << "\n";

    output = remappingMultiChannelImage(input, TransArry);
```

```
18.
19.     return cudaStatus;
20. }
```

La funzione `remappingMultiChannelImage()` ha il compito di calcolare il nuovo frame, OpenCV, mette a disposizione degli oggetti *cv::cuda::GpuMat.* Tali oggetti vanno a definire in modo trasparente all'utente, dell'area di memoria all'interno della GPU, tramite i metodi *upload (hostToDevice)* e *download (DeviceToHost)* con lo scopo di contenere una matrice di byte di terne RGB.

Ovviamente è possibile accedere a tale zona tramite a dei puntatori, definiti dalla libreria stessa.

```
1.  __global__ void new_remapping_kernel(cv::cuda::PtrStepSz<uchar3> src, int numRows, int num
    Cols, size_t step, int numChannel, int *tranfArray, cv::cuda::PtrStepSz<uchar3> out){
2.      int row = blockIdx.y * blockDim.y + threadIdx.y;
3.      int col = blockIdx.x * blockDim.x + threadIdx.x;
4.      int size = numRows * numCols;
5.      int idx;
6.      int homeX, homeY;
7.      int newhomeX, newhomeY;
8.      uchar3 pxval;
9.      idx = row * numCols + col;
10.     //if ((row < numRows) && (col < numCols))
11.     if (idx < numRows *numCols)
12.     {
13.         homeX=idx % numCols;
14.         homeY=idx / numCols;
15.         if(tranfArray[idx] != -1 ){
16.             newhomeX = tranfArray[idx] % numCols; // Col ID
17.             newhomeY = tranfArray[idx] / numCols;  // Row ID
18.             pxval = src(homeY, homeX );
19.             out(newhomeY, newhomeX ) = pxval;
20.         }
21.     }
22. }
23. cv::Mat remappingMultiChannelImage(Mat image, int *tranfArray){
24.     cudaError_t cudaStatus;
25.     dim3 blockDim(16, 16);
26.     dim3 gridDim(ceil((float)image.cols / blockDim.x), ceil((float)image.cols / blockDim.y
    ), 1);
27.     int num_RGBelem,size = image.rows * image.cols;
28.     Mat null_mat = Mat::zeros(cv::Size(image.rows, image.cols), CV_8UC3);
29.     cout << "Remapping image :\n \t \ttipo matrice :" << "CV_" + type2str(image.type()) <<
    endl;
30.     cv::Mat img;
31.     uchar *d_image, *d_output;
32.     int *d_tranfArray;
33.     //definisco l'immagine
34.     cv::cuda::GpuMat input, output;
35.     input.upload(image);
36.     output = input.clone();
37.     output.setTo(Scalar::all(0));
38.     //cout <<" \n alloco il vettore di transposizione \n";
39.     cudaStatus = cudaMalloc((void **) &d_tranfArray, sizeof(int) * size);
40.     if (cudaStatus != cudaSuccess) {
41.         fprintf(stderr, "cudaMalloc failed!");
42.         goto ErrorNewMultiRemapping;
43.     }
44.
45.     //cout <<" \n copio il vettore di transposizione \n";
46.     cudaStatus = cudaMemcpy(d_tranfArray,tranfArray,sizeof(int) * size, cudaMemcpyHostToDe
    vice);
```

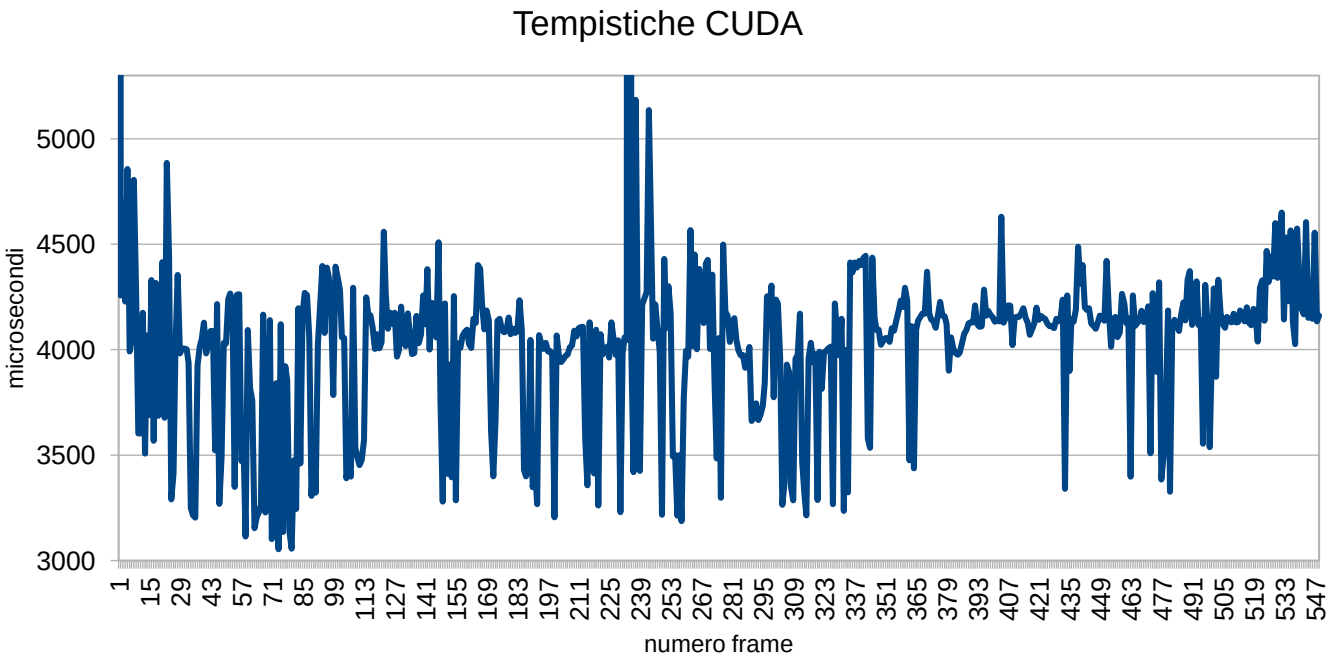
```
47.     if (cudaStatus != cudaSuccess) {
48.         fprintf(stderr, "CudaMemSetfailed: %s\n", cudaGetErrorString(cudaStatus));
49.         goto ErrorNewMultiRemapping;
50.     }
51.
52.     new_remapping_kernel<<<gridDim,blockDim>>> (input, input.rows, input.cols, input.step,
    image.channels(), d_tranfArray, output);
53.     cudaThreadSynchronize();
54.     output.download(img);
55.     return img;
56.
57. ErrorNewMultiRemapping:
58.     cudaFree(d_tranfArray);
59.     exit(0);
60. }
```

Anche in questa funzione, viene assegnato ad ogni pixel del frame un solo, thread, come si può vedere nella funzione *new_remapping_kernel()*

Conclusioni

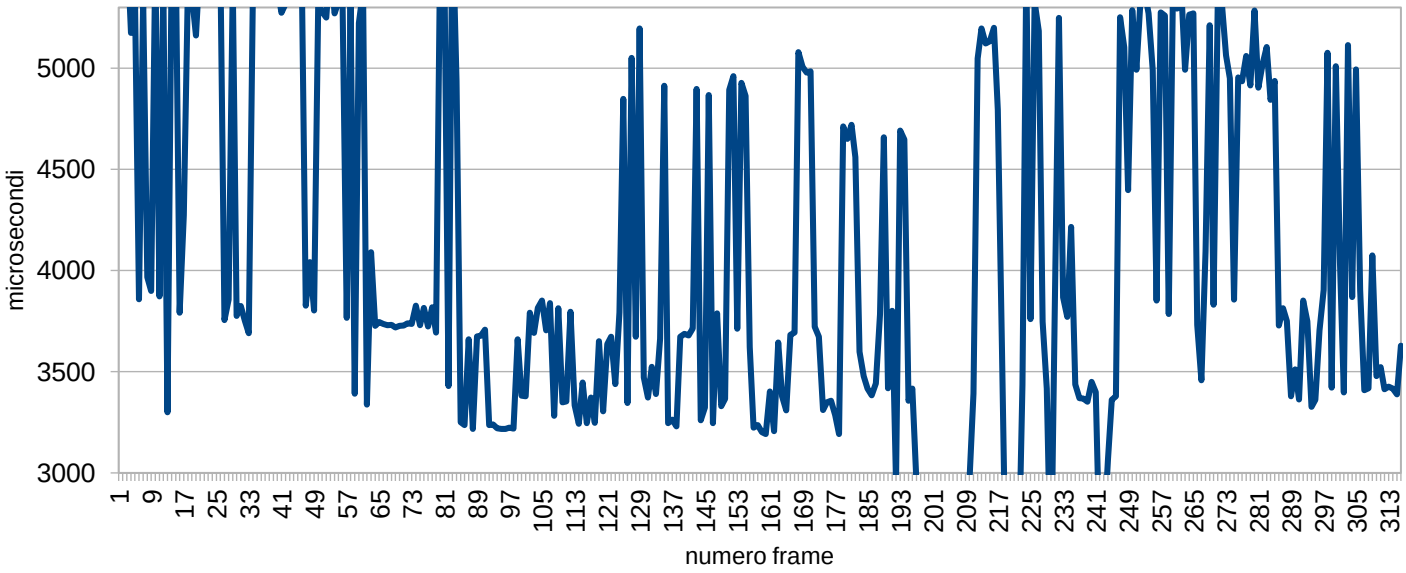
a. Confronto tra tempistiche

In questo grafico viene riportato l'andamento del tempo, in microsecondi, necessario per analizzare e trasporre un singolo frame tramite l'uso di cuda.



Per un singolo frame trascorrono in media 4163,72 µs

Per quanto riguarda l'analisi dello stream proveniente da webcam, tramite opencv si hanno i media 4132,18µs.



l'hardware usato per tali misurazioni è il seguente:

AMD® Ryzen 7 2700x eight-core processor × 16 thread @ 4.3GHz

GeForce GTX 750 Ti/PCIe/SSE2

16 GB Ram 3200Mhz dual channel

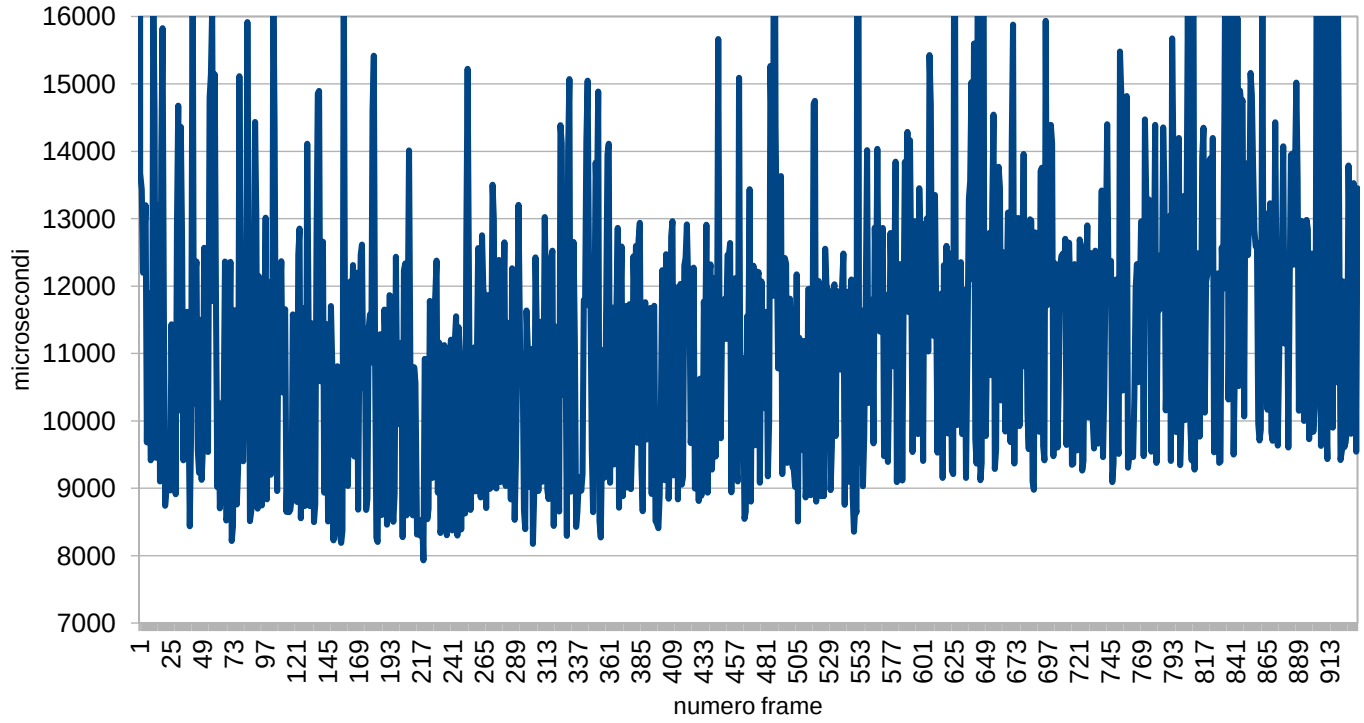
Analizzando sempre lo stream video proveniente da webcam, ma con un diverso hardware:

Intel® Core™ i5-6200U CPU @ 2.30GHz × 4

GeForce 920M/PCIe/SSE2

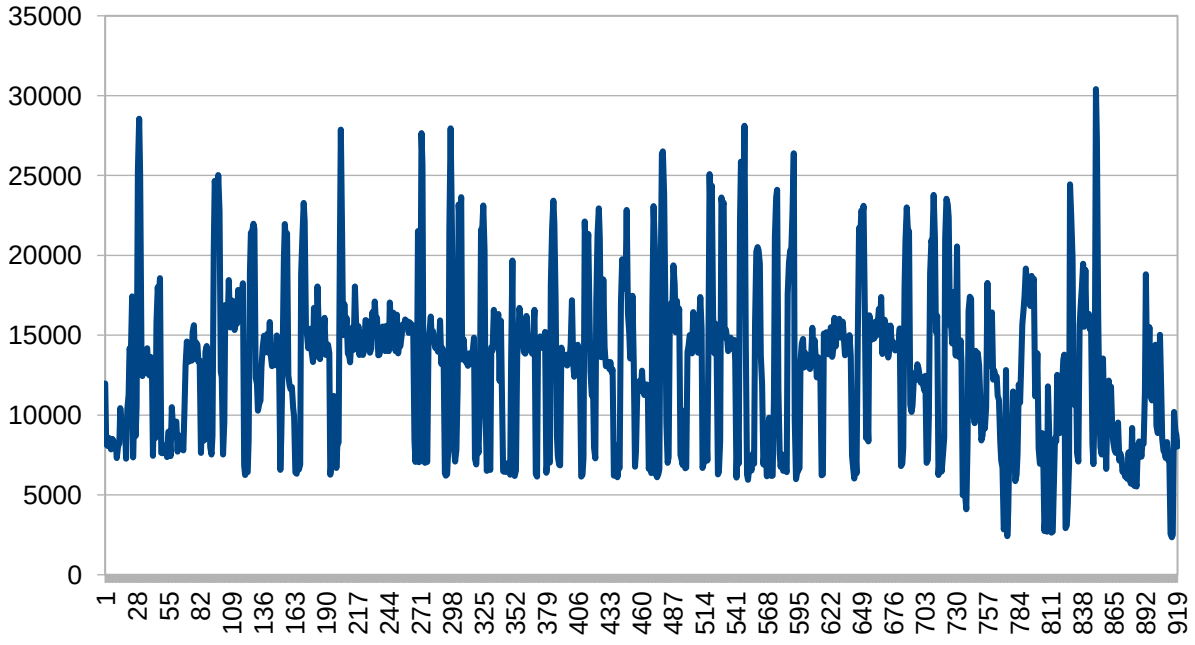
8 GB Ram

si ottengono i seguenti risultati:



media = 11427 µs

invece analizzando il frame tramite OpenCV:



media = 12952 µs

Il risultato più evidente è che l'architettura hardware incide moltissimo quelle performance dell'algoritmo, sia in versione parallelizzata che non. Inoltre, con un hardware meno reattivo, la versione parallelizzata, risulta essere mediamente più veloce. Questo risultato è importante poiché emerge la potenzialità della parallelizzazione.