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 | 255903@studenti.unimore.it | 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 omografica 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. Tale trasposizione deve essere effettuata nel più beve tempo possibile in modo da non creare troppo ritardo, ed un calo degli FPS.

Lo scopo progettuale è stato quello di implementare tale algoritmo, e fornirne due versioni diverse:

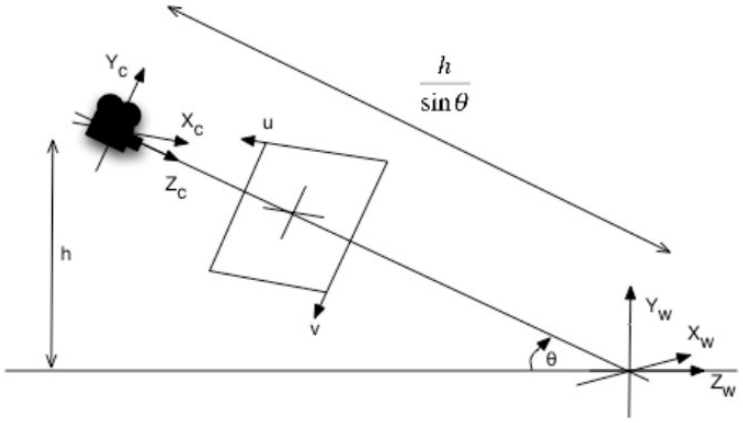
- una che utilizzi solamente funzioni già esistenti di OpenCV
- l’altra che implementi delle funzioni accelerate in CUDA programmate appositamente

Infine, si è eseguito tale programma, su diverse architetture e si è calcolato lo speed up

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 :

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. Ai fini progettuali, si è considerato di riproiettare un piano con la coordinata Y = 0, inquadrato da una fotocamera con orientamento fissato, e distanza variabile. Tale procedimento matematico fa parte della branca di “visione artificiale” che sta assumendo sempre più rilievo soprattutto in tutti gli strumenti di assistenza alla guida dei veicoli stradali.



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 :

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

Dove (p11… p34 è il risultato della moltiplicazione K T R) essendo interessati al piano della strada ($Y_w = 0$), la precedente formula diventa:

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

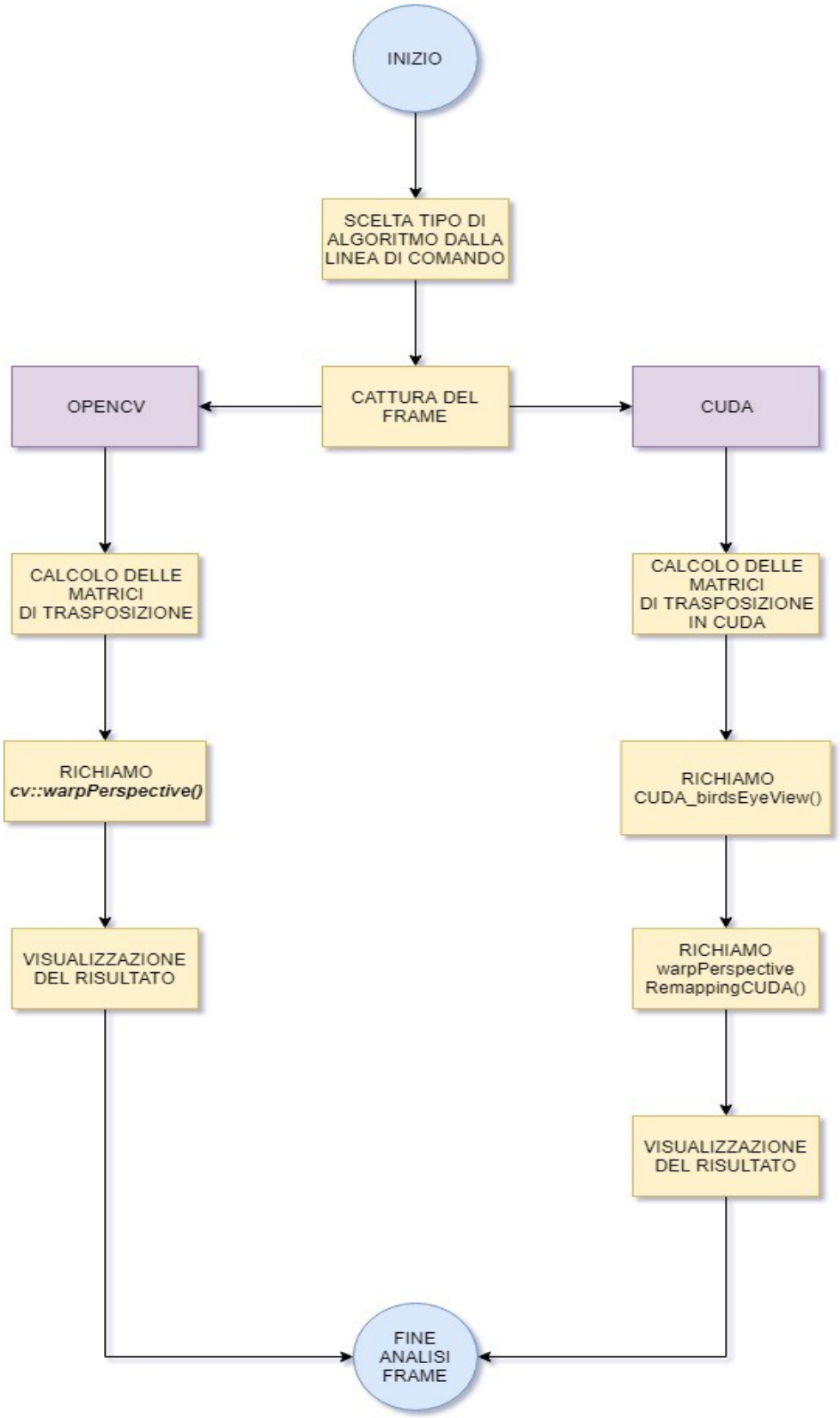
Grazie a quest'ultima equazione è possibile ottenere il mapping top down desiderato.

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/ ' | no
    thing > " << 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_cloc
    k::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::c
    hrono::microseconds>(end - begin).count() << "[µs]" << std::endl;
57.             std::cout << "Time difference = " << std::chrono::duration_cast<std::c
    hrono::nanoseconds> (end - begin).count() << "[ns]" << std::endl;
58.         }else{
59.             std::chrono::steady_clock::time_point begin = std::chrono::steady_cloc
    k::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::c
    hrono::microseconds>(end - begin).count() << "[µs]" << std::endl;
63.             std::cout << "Time difference = " << std::chrono::duration_cast<std::c
    hrono::nanoseconds> (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();
11.    double w = (double)input_size.width, h = (double)input_size.height;
12.
13.
14.    // Projecion matrix 2D -> 3D
15.    Mat A1 = (Mat_<float>(4, 3)<<
16.        1, 0, -w/2,
17.        0, 1, -h/2,
18.        0, 0, 0,
19.        0, 0, 1 );
20.
21.    // Rotation matrices Rx, Ry, Rz
22.    Mat RX = (Mat_<float>(4, 4) <<
23.        1, 0, 0, 0,
24.        0, cos(alpha), -sin(alpha), 0,
25.        0, sin(alpha), cos(alpha), 0,
26.        0, 0, 0, 1 );
27.
28.    Mat RY = (Mat_<float>(4, 4) <<
29.        cos(beta), 0, -sin(beta), 0,
30.        0, 1, 0, 0,
31.        sin(beta), 0, cos(beta), 0,
32.        0, 0, 0, 1  );
33.
34.    Mat RZ = (Mat_<float>(4, 4) <<
35.        cos(gamma), -sin(gamma), 0, 0,
36.        sin(gamma), cos(gamma), 0, 0,
37.        0, 0, 1, 0,
38.        0, 0, 0, 1  );
39.    // R - rotation matrix
40.    Mat R = RX * RY * RZ;
41.    // T - translation matrix
42.    Mat T = (Mat_<float>(4, 4) <<
43.        1, 0, 0, 0,
44.        0, 1, 0, 0,
45.        0, 0, 1, dist,
46.        0, 0, 0, 1);
47.    // K - intrinsic matrix
48.    Mat K = (Mat_<float>(3, 4) <<
49.        focalLength, 0, w/2, 0,
50.        0, focalLength, h/2, 0,
51.        0, 0, 1, 0
52.        );
53.    Mat transformationMat = K * (T * (R * A1));
54.    warpPerspective(input, output, transformationMat, input_size, INTER_CUBIC | WA
   RP_INVERSE_MAP);
55.    return;
56. }
```

Versione accelerata:

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] = {
```

```
        1, 0, -w/2,
        0, 1, -h/2,
        0, 0, 0,
        0, 0, 1
    };

    float RX[16] = {
        1, 0, 0, 0,
        0, cos(alpha), -sin(alpha), 0,
        0, sin(alpha), cos(alpha), 0,
        0, 0, 0, 1
    };

    float RY[16] ={
        cos(beta), 0, -sin(beta), 0,
        0, 1, 0, 0,
        sin(beta), 0, cos(beta), 0,
        0, 0, 0, 1
    };

    float RZ[16] = {
        cos(gamma), -sin(gamma), 0, 0,
        sin(gamma), cos(gamma), 0, 0,
        0, 0, 1, 0,
        0, 0, 0, 1
    };

    // cout << "stampo RX \n";
    // stampaMatrice(RX , 4, 4);
    // R - rotation matrix
    // Mat R = RX * RY * RZ;

    float R[16], XY[16];
    error = matrixMultiplication(RX, RY, XY, 4, 4, 4, 4);
    if (error != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        exit(0);
    }
    error = matrixMultiplication(XY, RZ, R, 4, 4, 4, 4);
    if (error != cudaSuccess) {
        fprintf(stderr, "cudaMalloc failed!");
        exit(0);
    }

    // T - translation matrix
    float T[16] = {
        1, 0, 0, 0,
        0, 1, 0, 0,
        0, 0, 1, dist,
        0, 0, 0, 1
    };
    // K - intrinsic matrix
    float K[12] = {
        focalLength, 0, w/2, 0,
        0, focalLength, h/2, 0,
        0, 0, 1, 0
    };

    //Mat transformationMat = K * (T * (R * A1));
    float R_A1[12], T_RA1[12], transformationvector[9];

    error = matrixMultiplication(R, A1, R_A1, 4, 4, 4, 3);
    if (error != cudaSuccess) {
        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*, che 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.

Vediamo in dettaglio il suo funzionamento, ogni thread, calcola la sua posizione all'interno del blocco della griglia. Ora data la sua posizione, si controlla prima se le coordinate sono all'interno delle dimensioni della matrice risultante, se si, ciascun thread scorre l'intera riga e l'intera colonna delle due matrici messe prodotto, basandosi sulla sua posizione.

```
1.  __global__ void generic_mat_mul(float *A, float *B, float *C, int numARows,int num
    AColumns, 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.
```

Per spiegare meglio il funzionamento, riporto ora un piccolo tracing del codice:

Dati *A[4] = {2,3,4,5}, B[4] = {6,7,8,9},* supponendo che *row = 0* e *col =0,* si ottiene:

*A[0] = 2 * B[0] = 6*

*A[1] = 3 * B[2] = 8*

C[0] = 36

invece se *row = 0* e *col = 1* si ha :

*A[0] = 2 * B[1] = 7*

*A[1] = 3 * B[3] = 9*

C[1] = 41

per le successive righe si avrà lo stesso procedimento.

Dalla porzione di tracing riportata precedentemente, si può vedere che ciascun thread va a scorrere tutta la riga della prima matrice e tutta la colonna della seconda e calcola le somme dei singoli prodotti.

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.

```
13. cudaError_t matrixMultiplication(float *A, float *B, float *C, int numARows,int nu
    mAColumns, 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!");
32.         goto Error;
33.     }
34.     //copio i vettori
35.     cudaMemcpy(d_A,A,sizeof(float)*numARows*numAColumns,cudaMemcpyHostToDevice);
36.     cudaMemcpy(d_B,B,sizeof(float)*numBRows*numBColumns,cudaMemcpyHostToDevice);
37.     cudaMemset(d_C, 0, numARows * numBColumns * sizeof(float));
38. 
```

```
39.     generic_mat_mul<<<gridDim, blockDim>>>(d_A, d_B, d_C, numARows, numAColumns, n
    umBRows, numBColumns);
40.     cudaThreadSynchronize();
41.     cudaStatus = cudaGetLastError();
42.     if (cudaStatus != cudaSuccess) {
43.         fprintf(stderr, "Kernel launch failed: %s\n", cudaGetErrorString(cudaStatu
    s));
44.         goto Error;
45.     }
46.
47.     cudaStatus = cudaDeviceSynchronize();
48.     if (cudaStatus != cudaSuccess) {
49.         fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launch
    ing addKernel!\n", cudaStatus);
50.         goto Error;
51.     }
52.
53.     cudaStatus = cudaMemcpy(C, d_C,numARows * numBColumns * sizeof(float), cudaMem
    cpyDeviceToHost);
54.     if (cudaStatus != cudaSuccess) {
55.         fprintf(stderr, "cudaMemcpy failed!");
56.         goto Error;
57.     }
58.     //@@ Free the GPU memory here
59. Error:
60.     cudaFree(d_A);
61.     cudaFree(d_B);
62.     cudaFree(d_C);
63.     return cudaStatus;
64.
65. }
```

warpPerspectiveRemappingCUDA:

Questa funzione è il nocciolo del progetto ed ha il compito di calcolare la traslazione del frame video. Tramite "*warpPerspectiveRemappingCUDA()*" 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), ed in più si va ad effettuare l'effettiva traslazione dei pixel. Ciò viene fatto in un unico *kernel CUDA*

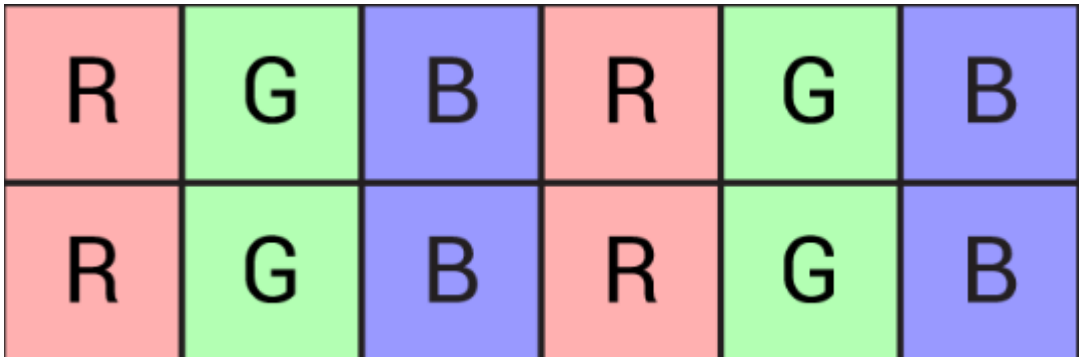
```
1. __global__ void pixelRemappingCudaKernel(cv::cuda::PtrStepSz<uchar3> src,
2.                                          cv::cuda::PtrStepSz<uchar3> out,
3.                                          size_t step,
4.                                          int numChannel,
5.                                          float *H,
6.                                          int *transfArray,
7.                                          int numRows,
8.                                          int numCols){
9.
10.   int idx = blockIdx.x * blockDim.x + threadIdx.x;
11.
12.   int MaxX,MaxY = -1000;
13.   int MinX,MinY =  1000;
14.   uchar3 pxval;
15.   int homeX, homeY;
16.   int newhomeX, newhomeY;
17.
```

```
  if (idx < numRows * numCols) {
    homeX=idx % numCols;
    homeY=idx / numCols;

    float x  = (H[0] * (homeX)) +( H[1] * (homeY)) +  H[2] ;
    float y  = (H[3] * (homeX)) +( H[4] * (homeY)) +  H[5] ;
    float s  = (H[6] * (homeX)) +( H[7] * (homeY)) +  H[8] ;

    x = floor(x/s);

    y = floor(y/s);

    // for the first col in TransMatrix
    if (homeX == 0){
      if (x > MaxX) MaxX = x;
      if (x < MinX) MinX = x;
    }

    //for thee first row in TransMatrix
    if (homeY == 0){
      if (y > MaxY) MaxY = y;
      if (y < MinY) MinY = y;
    }
    if( y >= numRows || y<0 || x >= numCols ||  x < 0){
      transfArray[idx]  = -1;
    }else{
      transfArray[idx] = (y * numCols + x);

      //------pezzo aggiunto

      homeX=idx % numCols;
      homeY=idx / numCols;
      newhomeX = transfArray[idx] % numCols; // Col ID
      newhomeY = transfArray[idx] / numCols;  // Row ID
      //srcval = src(homeY, homeX*numChannel);

      pxval = src(homeY, homeX );
      out(newhomeY, newhomeX ) = pxval;
    }

  }

}


cv::cuda::GpuMat input, output;
cudaError_t warpPerspectiveRemappingCUDA(Mat inputFrame, Mat &outputFrame, Mat
H){
  cudaError_t cudaStatus;
  int size = inputFrame.rows * inputFrame.cols;
  int channels   = input.channels();
  int *TransArry = (int *)malloc(sizeof(int)*size);
  float *vecH = (float *)malloc(sizeof(float) * H.rows * H.cols);
  float *d_H;
  int *d_T;
  matToArray(vecH, H, H.rows, H.cols);

  // ALLOCO LA MEMORIA sulla GPU
```

```
75.
76.
77.   cudaStatus = cudaMalloc((void **) &d_H, sizeof(float)*H.rows * H.cols);
78.   if (cudaStatus != cudaSuccess) {
79.     fprintf(stderr, "cudaMalloc failed!");
80.     goto ErrorWarp;
81.   }
82.
83.   cudaStatus = cudaMalloc((void **) &d_T, sizeof(int) * size);
84.   if (cudaStatus != cudaSuccess) {
85.     fprintf(stderr, "cudaMalloc failed!");
86.     goto ErrorWarp;
87.   }
88.
89.   // COPIO I DATI SULLA GPU
90.
91.   //copio sul device la matrice H
92.   cudaStatus = cudaMemcpy(d_H,vecH,sizeof(float)*H.rows * H.cols,cudaMemcpyHost
ToDevice);
93.   if (cudaStatus != cudaSuccess) {
94.     fprintf(stderr, "CudaMemCpy failed: %s\n", cudaGetErrorString(cudaStatus))
;
95.     goto ErrorWarp;
96.   }
97.
98.   //copio sul device lo spazio per il vettor di trasposizione
99.   cudaStatus = cudaMemset(d_T, 0, sizeof(int) * size);
100.   if (cudaStatus != cudaSuccess) {
101.     fprintf(stderr, "CudaMemSetfailed: %s\n", cudaGetErrorString(cudaStatus))
;
102.     goto ErrorWarp;
103.   }
104.
105.   //carico il frame in input sulla GPU
106.
107.   input.upload(inputFrame);
108.   //output.create(cv::Size(image.rows, image.cols), CV_8UC3);
109.   output = input.clone();
110.   output.setTo(Scalar::all(0));
111.
112.   pixelRemappingCudaKernel<<<ceil(size/1024.0),1024>>>(input, output, inputFr
ame.step,
113.   inputFrame.channels(), d_H, d_T, inputFrame.rows, inputFrame.cols);
114.   cudaDeviceSynchronize();
115.   cudaStatus = cudaGetLastError();
116.   if (cudaStatus != cudaSuccess) {
117.     fprintf(stderr, "Kernel launch failed: %s\n", cudaGetErrorString(cudaStat
us));
118.     goto ErrorWarp;
119.   }
120.
121.   output.download(outputFrame);
122.
123.   cudaFree(d_H);
124.   cudaFree(d_T);
125.   return cudaStatus;
126.
127. ErrorWarp:
```

```
128.    cudaFree(d_H);
129.    cudaFree(d_T);
130.    return cudaStatus;
131.}
```

La funzione appena riportata, ha il compito di effettuare la traslazione lineare della vecchia posizione del pixel in quella nuova.
La nuova posizione è data dal vettore *tranfArray*, che avrà tante posizioni quanti sono i pixel dell'immagine.
L'immagine RGB analizzata ha i canali interallacciati, ciò significa che la matrice è così fatta:



ciascun thread, che ne sono tanti quanti i pixel dell'immagine, calcola la nuova posizione del pixel, inserendola nel vettore "*tranfArray",* una volta terminato il calcolo preleva la tripletta *RGB,* e la trascrive nella nuova posizione.
Per trovare tale mapping si applica la seguente formula

$$\begin{bmatrix} x'_{i_1} \\ x'_{i_2} \\ x'_{i_3} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \cdot \begin{bmatrix} x_{i_1} \\ x_{i_2} \\ x_{i_3} \end{bmatrix}$$

dove gli X'_i sono le nuove coordinate, mentre gli X_i, sono le vecchie posizioni.
Sviluppando la precedente formula si ha :

$$\begin{cases} x'_{i_1} = h_{11}x_{i_1} + h_{12}x_{i_2} + h_{13}x_{i_3} \\ x'_{i_2} = h_{21}x_{i_1} + h_{22}x_{i_2} + h_{23}x_{i_3} \\ x'_{i_3} = h_{31}x_{i_1} + h_{32}x_{i_2} + h_{33}x_{i_3} \end{cases}$$

Risolvendo tale sistema si ottiene solamente 3 equazioni non dipendenti. Infatti i parametri h_{11} .. h_{33} non devono essere considerati singolarmente, ma va considerato il loro rapporto tra essi, ottenendo così solamente 8 gradi di libertà.
Ottenendo:

$$\begin{cases} x'_{i_1} \cdot (h_{21}x_{i_1} + h_{22}x_{i_2} + h_{23}x_{i_3}) = (h_{11}x_{i_1} + h_{12}x_{i_2} + h_{13}x_{i_3}) \cdot x'_{i_2} \\ x'_{i_2} \cdot (h_{31}x_{i_1} + h_{32}x_{i_2} + h_{33}x_{i_3}) = (h_{21}x_{i_1} + h_{22}x_{i_2} + h_{23}x_{i_3}) \cdot x'_{i_3} \\ x'_{i_3} \cdot (h_{11}x_{i_1} + h_{12}x_{i_2} + h_{13}x_{i_3}) = (h_{31}x_{i_1} + h_{32}x_{i_2} + h_{33}x_{i_3}) \cdot x'_{i_1} \end{cases}$$

Una volta che sono state definite le equazioni che descrivono il cambio di coordinate, le nuove potranno essere ricavate tramite le seguenti equazioni:

$$x'_{i1} = \frac{h_{1,0} + h_{1,1}x_{i1} + \cdots + h_{1,n}x_{in}}{h_{0,0} + h_{0,1}x_{i1} + \cdots + h_{0,n}x_{in}}$$

$$\vdots$$

$$x'_{in} = \frac{h_{n,0} + h_{n,1}x_{i1} + \cdots + h_{n,n}x_{in}}{h_{0,0} + h_{0,1}x_{i1} + \cdots + h_{0,n}x_{in}}$$

VERSIONE NON ACCELERATA:

Dell'algoritmo appena presentato, ne è stata effettuata una versione che non utilizza l'accellerazione in GPU, questo perché le chiamate di funzione di OpenCV effettuano delle operazioni leggermente diverse. Infatti l'algoritmo alternativo non effettua l'interpolazione dei pixel nelle posizioni in cui non vi è niente, perciò potrebbe non essere molto esplicativo confrontarlo con la chiamata di OpenCV.
In questo modo si può calcolare in modo più veritiero lo *speedup,* ed avere quindi una visuale effettiva del miglioramento di performance dello stesso algoritmo eseguendolo in GPU anziché in CPU

```
1.Mat tranImg;
2.Mat warpPerspectiveCPU(Mat A, Mat H){
3.   // allocate array of all locations
4.   int Numrows = A.rows;
5.   int Numcols = A.cols;
6.   int channels   = A.channels();
7.   int size = Numrows*Numcols;
8.   int MaxX,MaxY = -1000;
9.   int MinX,MinY =  1000;
10.   int *TransArry = (int *)malloc(sizeof(int)*size);
11.   int Idx;
12.   int homeX, homeY;
13.   A.copyTo(tranImg);
14.   tranImg = tranImg - tranImg;
15.   for (Idx=0; Idx < size; ++Idx ){
16.      homeX=Idx % Numcols;
17.      homeY=Idx / Numcols;
18.      float x  = (H.at<float>(0,0) * (homeX)) +( H.at<float>(0,1) * (homeY)) +
H.at<float>(0,2) ;
19.      float y  = (H.at<float>(1,0) * (homeX)) +( H.at<float>(1,1) * (homeY)) +
H.at<float>(1,2) ;
20.      float s  = (H.at<float>(2,0) * (homeX)) +( H.at<float>(2,1) * (homeY)) +
H.at<float>(2,2) ;
21.      x = floor(x/s);
22.      y = floor(y/s);
23.      if (homeX ==0){
24.         if (x > MaxX) MaxX = x;
25.         if (x < MinX) MinX = x;
26.      }
27.      if (homeY ==0){
28.         if (y > MaxY) MaxY = y;
29.         if (y < MinY) MinY = y;
30.      }
31.      if( y >= A.rows || y < 0 || x >= A.cols || x < 0){
32.         TransArry[Idx]  = -1;
33.      }else{
34.         TransArry[Idx] = (y * Numcols + x);
35.         homeX=Idx % Numcols;
36.         homeY=Idx / Numcols;
37.         int newhomeX=TransArry[Idx] % Numcols; // Col ID
38.         int newhomeY=TransArry[Idx] / Numcols;  // Row ID
39.         tranImg.at<uchar>(newhomeY, (newhomeX*channels)) = A.at<uchar>(homeY, ho
meX*channels);
40.         if(channels>1)
41.            tranImg.at<uchar>(newhomeY, newhomeX*channels+1) = A.at<uchar>(homeY,
homeX*channels+1);
42.         if(channels>2)
43.            tranImg.at<uchar>(newhomeY, newhomeX*channels+2) = A.at<uchar>(homeY,
homeX*channels+2);
44.      }
45.   }
46.   return tranImg;
47. }
```

Conclusioni

a. Benchmarking e conclusioni finali

L'algoritmo sopra presentato, è stato eseguito in tre diversi dispositivi, aventi capacita computazionali diverse.
In particolare sono stati utilizzati :

i. *AMD® Ryzen 7 2700x eight-core processor × 16 thread @ 4.3GHz*
   GeForce GTX 750 2GB Ti/PCIe/SSE2
   16 GB Ram 3200Mhz dual channel

ii. *Intel® Core™ i5-6200U CPU @ 2.30GHz × 4*
   GeForce 920M 2GB /PCIe/SSE2
   8 GB Ram 1600Mhz dual channell

iii. *Nvidia Jetson nano : 64-bit Quad-core ARM A57 @ 1.43GHz*
   128-core NVIDIA Maxwell @ 921MHz
   4GB 64-bit LPDDR4 @ 1600MHz | 25.6 GB/s

L'esecuzione dello stesso algoritmo, con parametri fissati, su dispositivi diversi, ha lo scopo di mettere in evidenza le diverse capacità computazionali dell'hardware e quindi capire in che modo e quali parti della computazione impattano di più sulle performance.
Per capire quando una versione è *"migliore"* rispetto ad un altra si è scelto di calcolare lo *SpeedUP*.
Esso è definito in questo modo:

$$SpeedUp = \frac{MediaTempiOpenCV[\mu s]}{MediaTempiCUDA[\mu s]}$$

Il risultato, che sarà adimensionale, potrà essere :

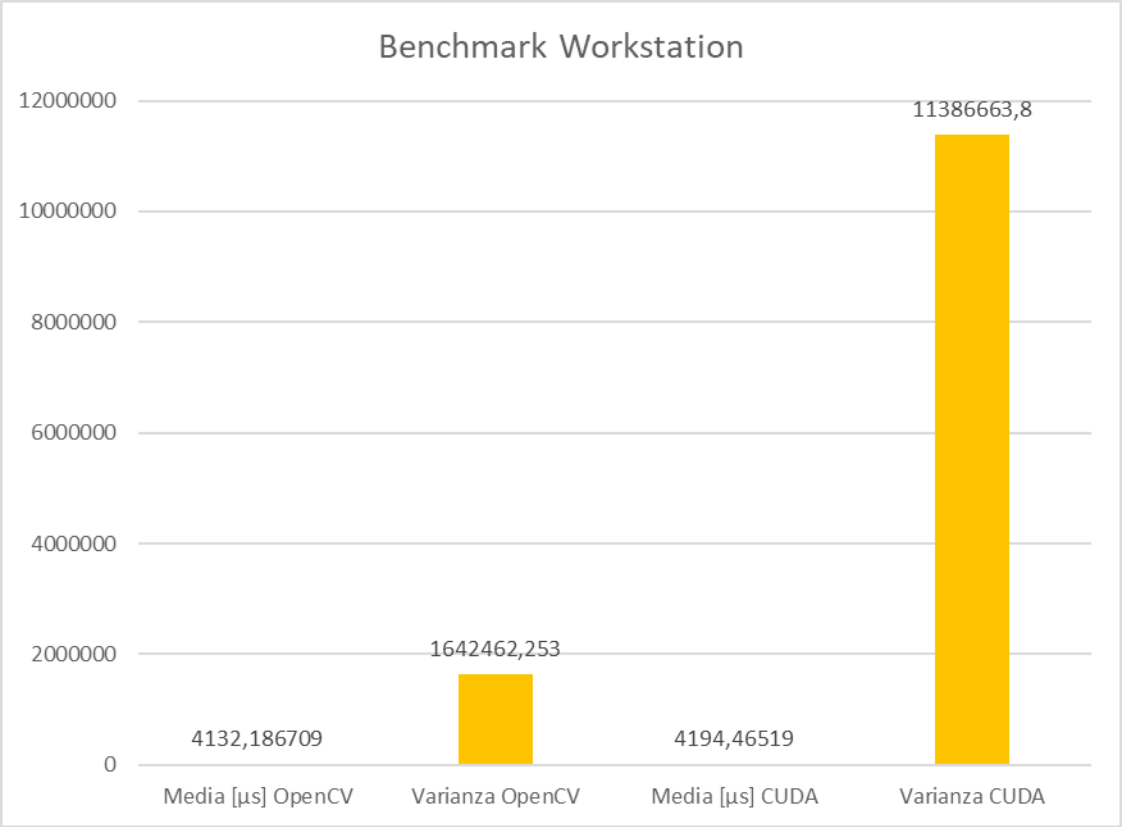
- *Maggiore di 1 :* ciò significa che il numeratore è più grande del denominatore e quindi la versione in CUDA risulterà essere più performante rispetto a quella OpenCV

- *Minore di 1 :* ciò significa che il numeratore è più piccolo del denominatore e quindi la versione in OpenCV risulterà più performante rispetto a quella CUDA

I benchmark riportati qui di seguito fanno riferimento all'analisi di un video, allegato al progetto, con parametri di riproiezione fissati, in modo da non ottenere una forte distorsione, poiché ciò comporterebbe un'elevata quantità di interpolazione dei pixel. Poiché l'algoritmo CUDA non si sofferma su tale tematica, il confronto potrebbe essere poco esplicativo, quindi si è scelta una configurazione iniziale di bassa o scarsa interpolazione.
Per avere una più ampia conoscenza delle varie tempistiche che le varie parti del codice CUDA impiegano ad essere eseguite, si è scelto di misurare il tempo delle *cudamalloc e cudamemcpy*.
In questo modo è stato possibile misurare l'impatto di queste operazioni e capire l'ipotetico guadagno eliminandole dal codice eseguito.
Per farlo si è scelto di utilizzare la libreria C++ "chrono" che permette di calcolare il tempo trascorso tra due punti del programma misurato in microsecondi.

Nei vari benchmark effettuati, saranno anche esposti dei grafici in cui sono stati messi a confronto i tempi delle due varianti dello stesso algoritmo. In essi viene riportato media e varianza dei tempi calcolati ed anche un grafico a torta in cui è possibile evincere quali parti dell'algoritmo CUDA impattano di più sulle performance

1) BENCHMARK - WORKSTATION



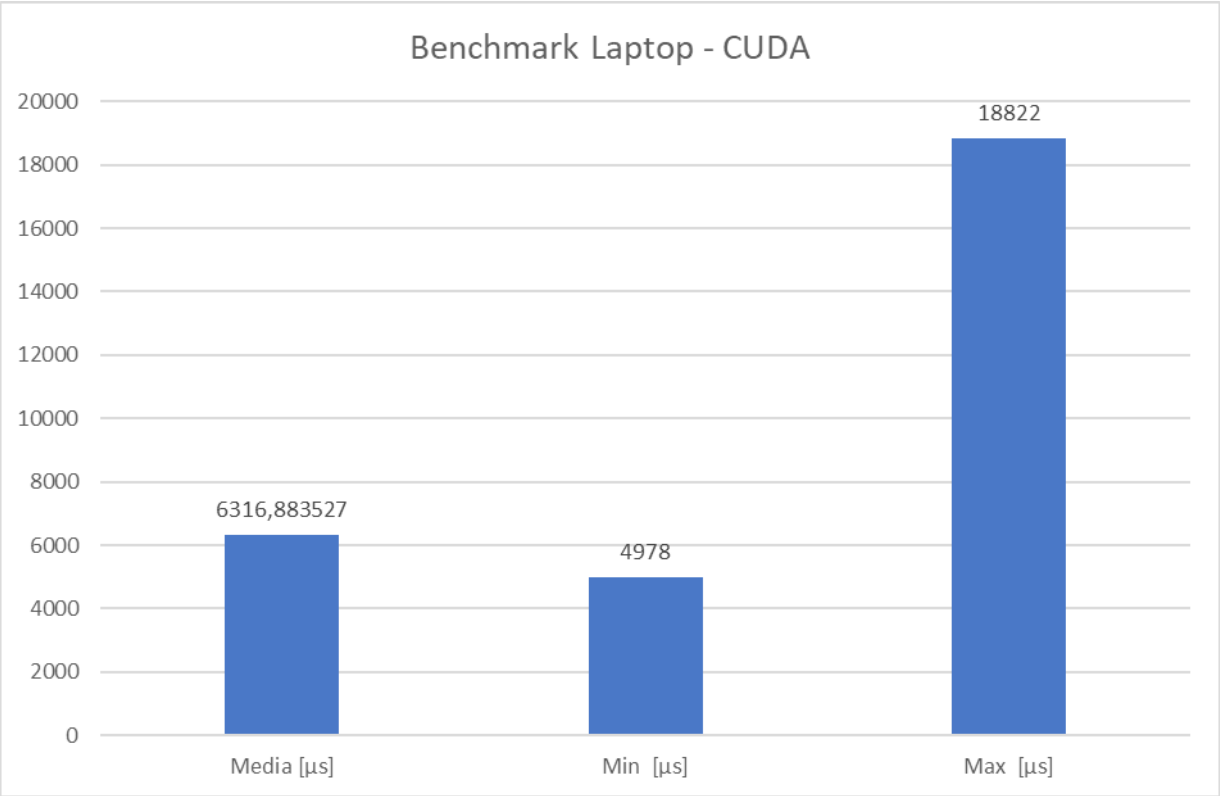
Dal grafico si può notare la media in µs dei tempi di esecuzione e la varianza. Minore e' la varianza e maggiore e' la concentrazione dei dati attorno al valore medio, mentre, Maggiore e' la varianza e maggiore e' la dispersione dei dati attorno al valore medio.

Ottenendo uno *SpeedUP = 0,98515222*

Ciò significa che la versione dell'algoritmo che sfrutta solamente chiamate della libreria OpenCV risulta essere leggermente più veloce, su quest'architettura.

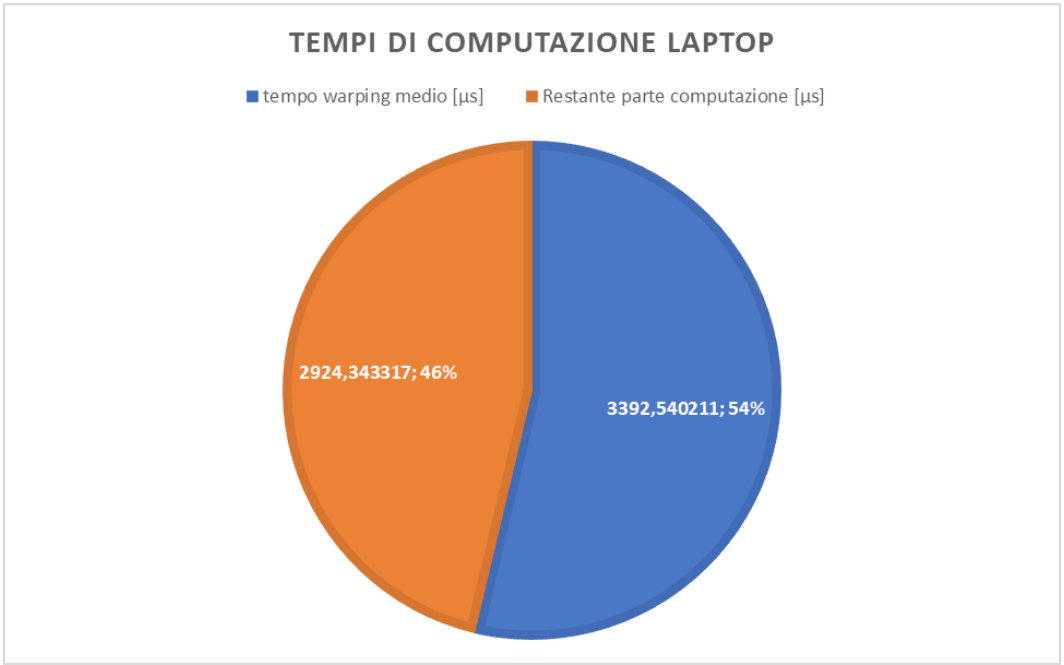
2) BENCHMARK - LAPTOP

Testando l'algoritmo in esame con il laptop si ottengono i seguenti risultati,



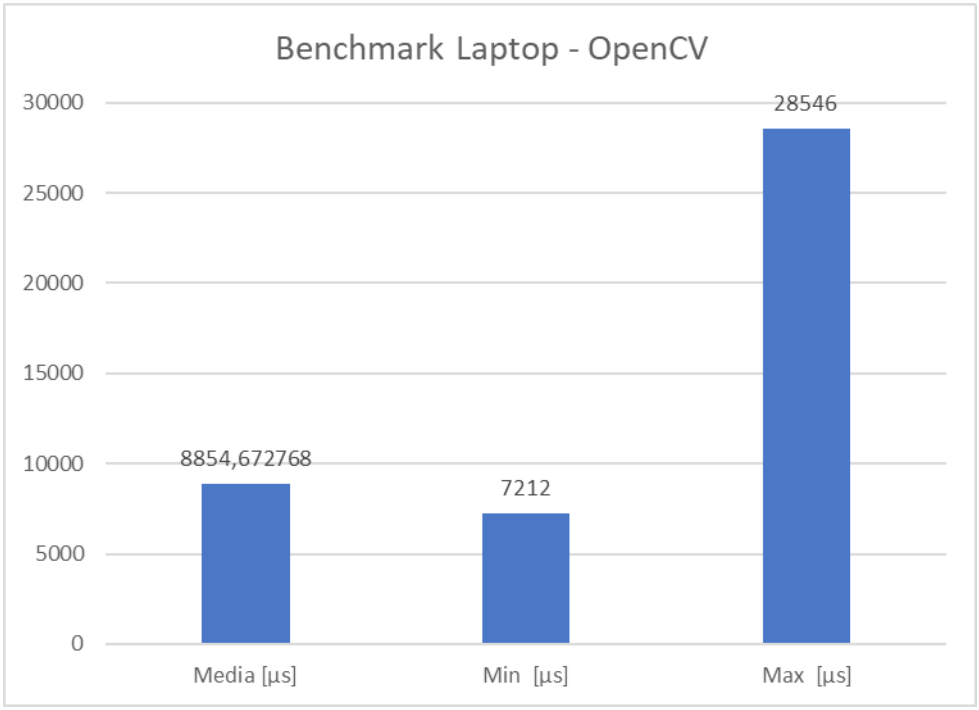
Il grafico appena riportato esprime il tempo medio, minimo e massimo in µs, che si impiega per analizzare un frame del video preso in analisi.

È interessante, inoltre, capire invece qual'è la porzione di tempo veramente utilizzata, sul totale, dall'algoritmo per effettuare la trasposizione del pixel, questo lo si può evincere dal seguente grafico a torta



Come riportato sopra, su una media di circa 6 mila µs, per il 54% del tempo, l'algoritmo effettua la trasposizione, nel restante 46% calcola le matrici di traslazione ed effettua gli swap di memoria.

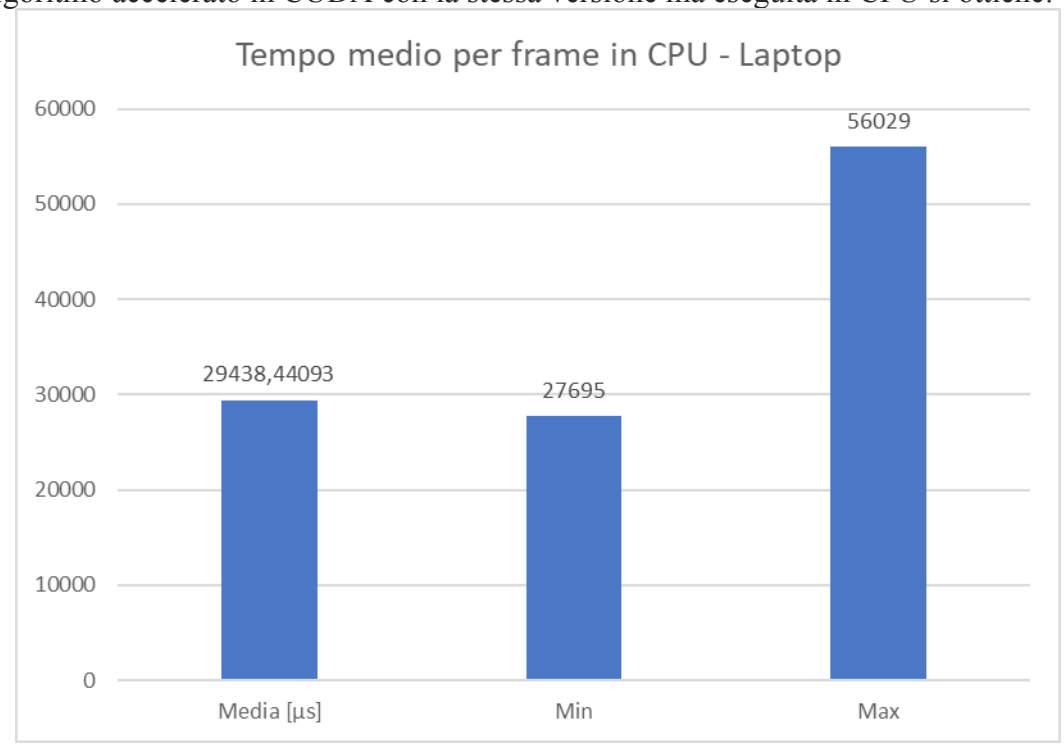
I tempi medi di esecuzione della variante utilizzante le chiamate messe a disposizione da OpenCV sono riportati sul seguente grafico



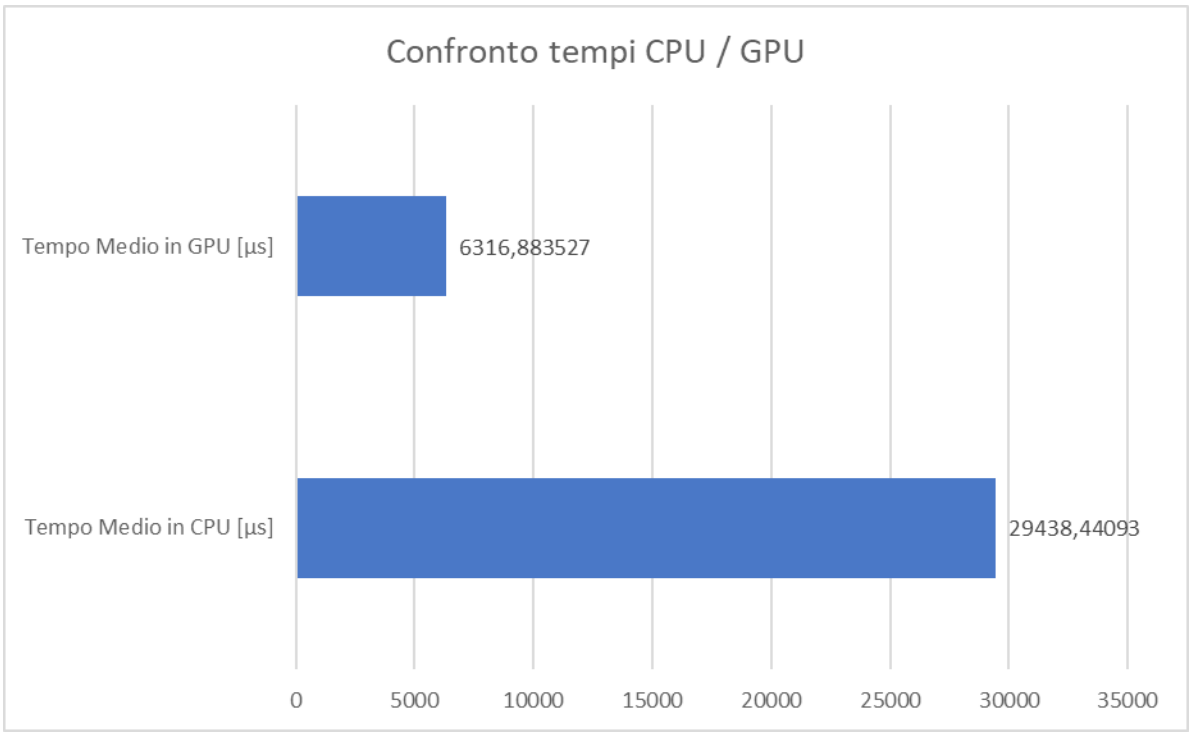
Calcolando anche in questo caso lo SpeedUp, come visto prima, si ottiene:

SpeedUP = 1,4018

Da ciò deriva che la versione parallelizzata risulta essere di molto migliore rispetto a quella utilizzante solo direttive OpenCV.
Per capire però in modo più appurato il vero guadagno in termini di performance, confrontando lo stesso algoritmo accelerato in CUDA con la stessa versione ma eseguita in CPU si ottiene:



Confrontando il tempo medio per frame sia in GPU che in CPU,si ottiene il seguente riscontro:



Calcolando lo SpeedUP come :

$$SpeedUp = \frac{tempiCPU}{tempiGPU}$$

si ottiene che, *SpeedUp = 4,6608*

b. Benchmarking su Nvidia Jetson Nano®

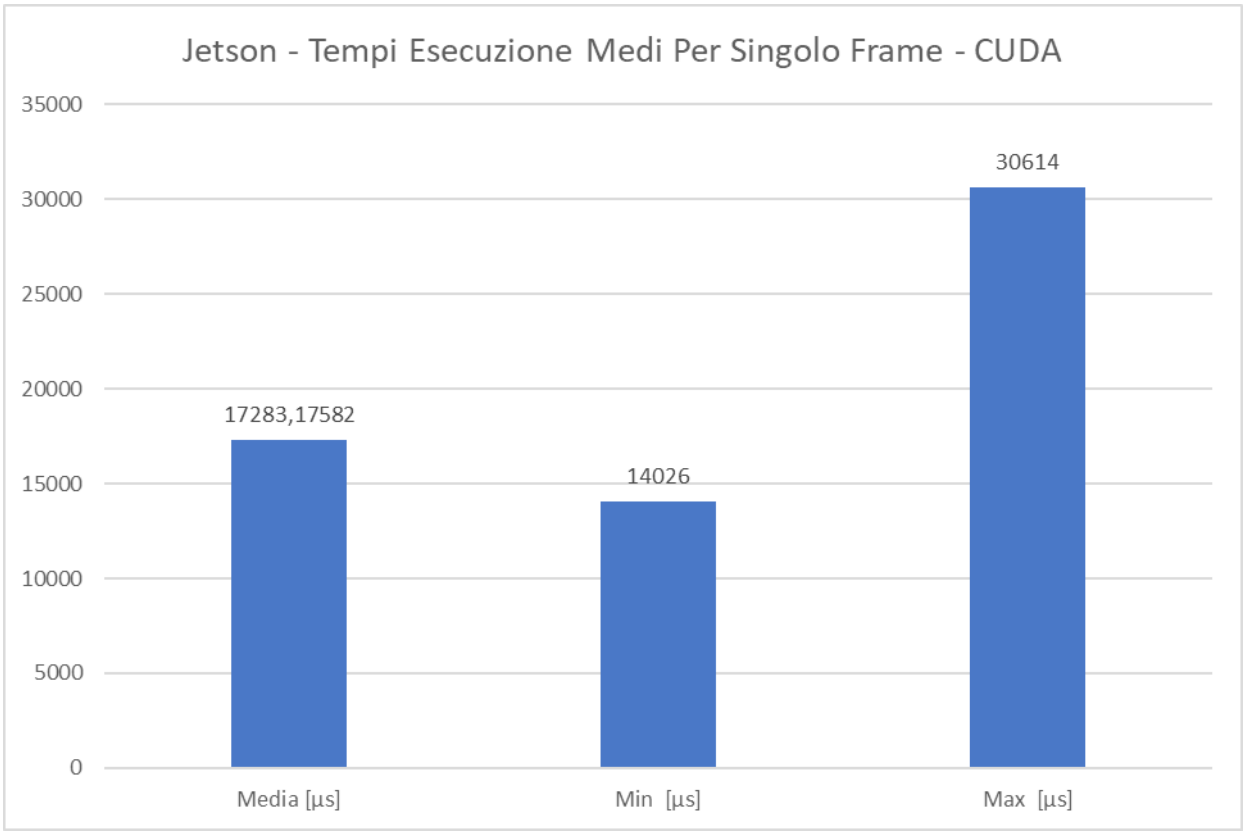
In seguito sono riportati i test di esecuzione, svolti sull'architettura embedded, prodotta da Nvidia, chiamata Jetson Nano.
Tale benchmark è molto interessante poiché, trattandosi di un'architettura embedded, essendo la memoria ram è condivisa sia dal processore che dalla GPU, non si ha più il collo di bottiglia introdotto dal bus PCI.
Inoltre proprio perché tale benchmark è stato effettuato con un'architettura embedded, per prelevare le tempistiche si è scelto di non aprire e chiudere un file di testo, ma bensì, le *cout* dei tempi anziché essere state mandate a video nel terminale sono state inserite in un file col seguente comando shel unix:

```
$ ./app y <video path> > tempi.txt
```

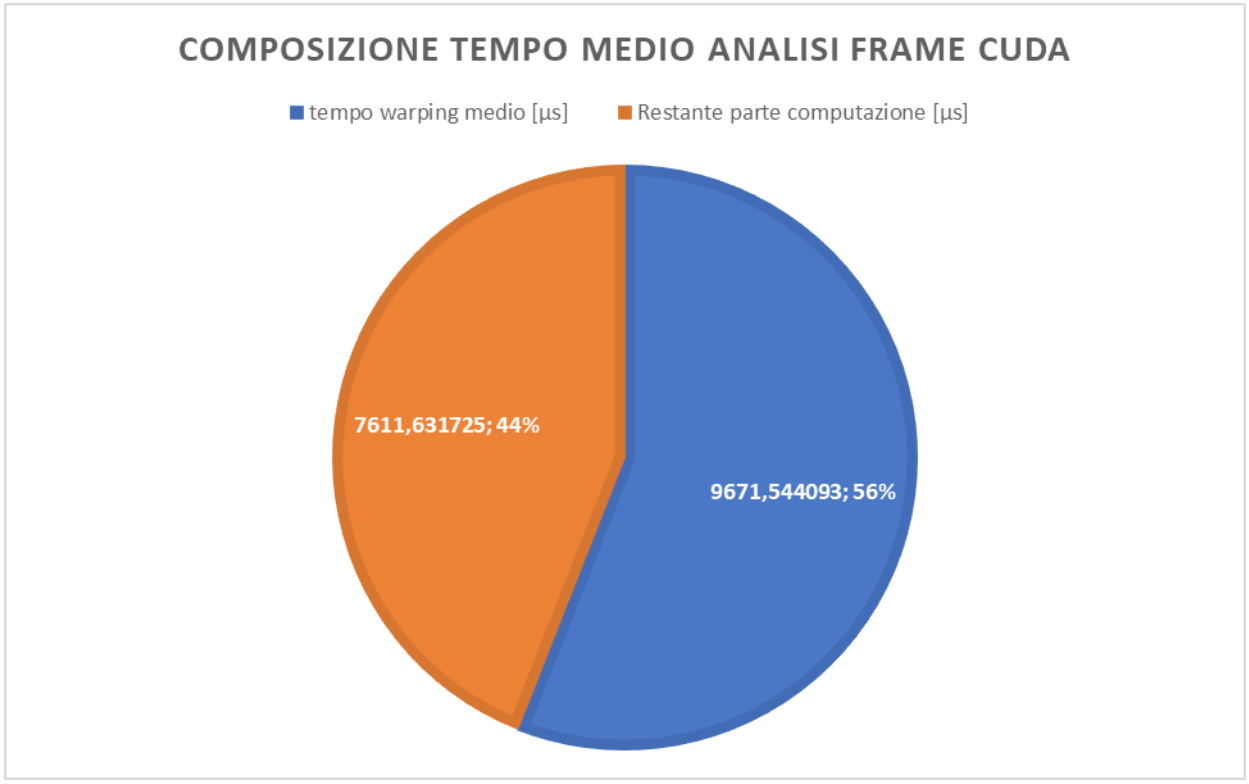
In questo modo si evitano delle aleatorietà che possono essere introdotte dall'apertura e chiusura degli stream durante l'esecuzione del programma.

Inoltre, solo per tale architettura, per capire in maniera più approfondita, ed effettuare un confronto migliore, oltre alle differenze tra la versione sfruttante le direttive di OpenCV e CUDA, si è scelto di confrontare i tempi tra l'algoritmo CUDA e la sua stessa versione ma eseguita in CPU.

Questo si è reso necessario perché la versione di OpenCV e quella scritta ai fini progettuali, non sono del tutto confrontabili, come visto precedentemente.

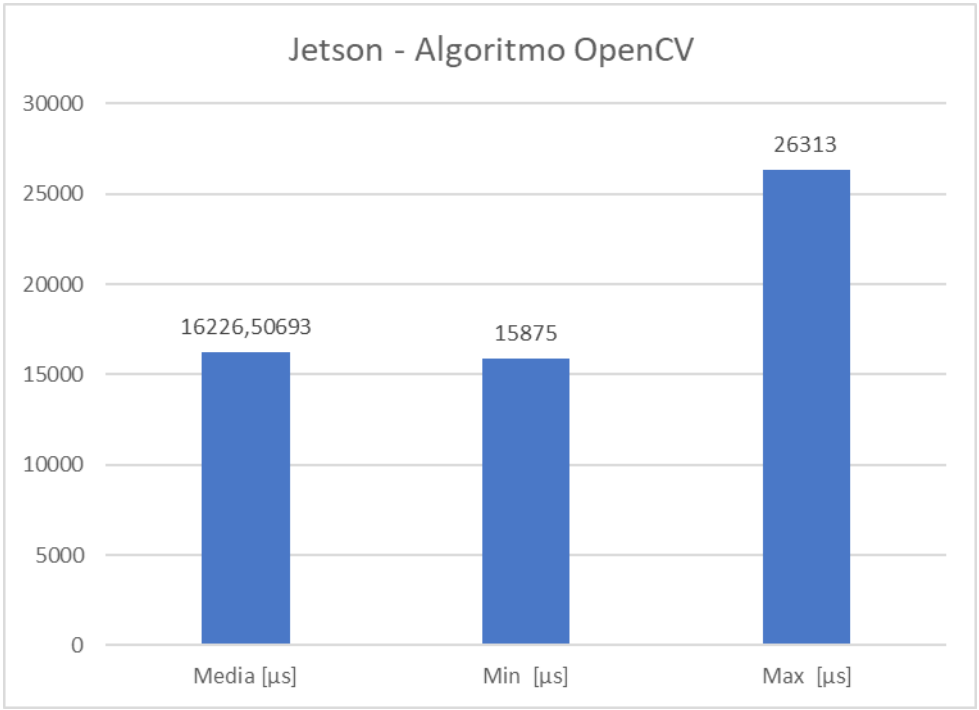


Anche in questo caso è interessante capire la ripartizione dei tempi, cioè dato il tempo medio per un frame, quanta percentuale è utilizzata per effettuare il remapping, ciò è dato dal seguente grafico a torta:



In questo caso, il 56% del tempo medio necessario ad analizzare un frame, viene utilizzato per effettuare il remapping dei pixel, nel restante 44% viene invece calcolato le matrici di traslazione e le restanti operazioni.

L'algoritmo sfruttante le direttive OpenCV impiega invece:

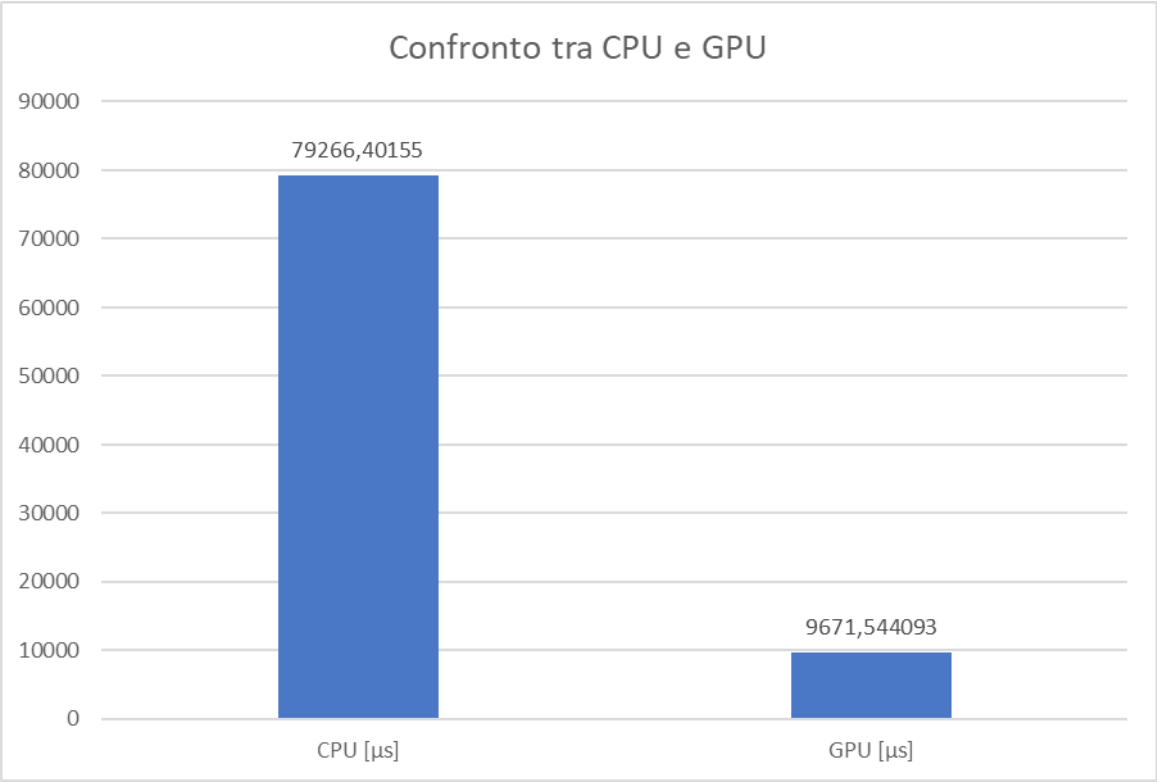


Misurando lo speedUP si ottiene:

SpeedUP = 0,9388

Questo risultato porta ad osservare che, l'algoritmo che sfrutta i kernel CUDA peggiora le performace di poco, rispetto alla versione OpenCV.

Il risultato però più sorprendente è invece il confronto tra lo stesso algoritmo eseguito solamente in CPU con quello accelerato in CUDA, infatti si ottiene il seguente grafico:



calcolando lo *speedup* come

$$SpeedUp = \frac{tempiCPU}{tempiGPU}$$

si ottiene:

SpeedUp = 8,1962

Tale risultato è molto sorprendente, poiché si migliora di circa l'80%, e questo certifica come effettuare una corretta gestione della memoria, ed un corretto uso della cosiddetta *offloading programming model* aiuti di molto a migliorare le performance di un algortimo, soprattutto su architetture embedded.

c. Ottimizzazioni Finali

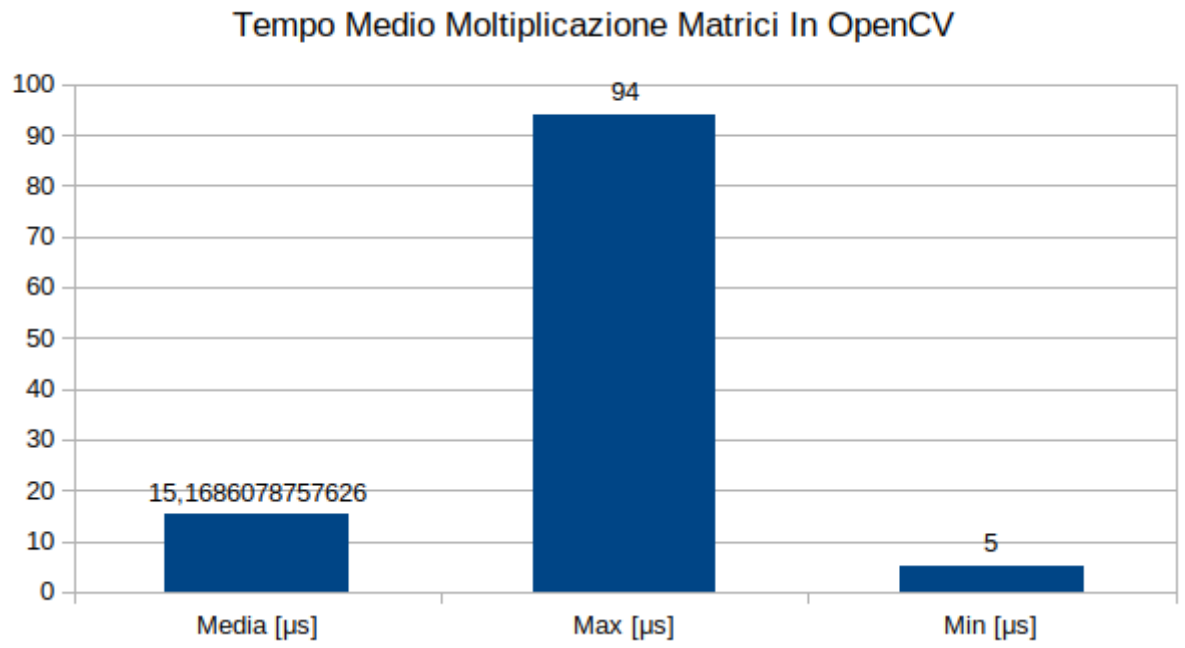
Dai risultati presentati sopra, e dai rispettivi grafici a torta, è possibile osservare che per effettuare la traslazione dell'immagine si necessita in media solo il 60% di tempo sul totale della media di computazione CUDA. Il restante 40%, circa, invece è utilizzato per fare altre operazioni non inerenti al "*warping*" dell'immagine.
In questo 40%, circa, di tempo speso, vi è contenuto, sia il calcolo delle matrici di traslazione e sia i vari swap di memoria necessari per effettuare l'offload di queste operazioni.
Ma osservando che le matrici che si inviano alla GPU sono molto piccole, al massimo si hanno delle moltiplicazioni tra matrici 4 x 4, viene meno il vantaggio di delegare tale calcolo alla GPU, poiché il tempo degli swap di memoria tra CPU e GPU, sarà sicuramente maggiore rispetto al calcolo effettivo della moltiplicazione.
Da ciò ne consegue che è stata scritta una nuova funzione che, effettua il calcolo delle matrici di traslazione in CPU, tramite l'overload dell'operatore * fornito da OpenCV, e che calcola la riproiezione dell'immagine in cuda.

```
void CUDA_birdsEyeView_OPT(const Mat &input, Mat &output){
    double focalLength, dist, alpha, beta, gamma;

    alpha =((double)alpha_ -90) * PI/180;
    beta =((double)beta_ -90) * PI/180;
    gamma =((double)gamma_ -90) * PI/180;
    focalLength = (double)f_;
    dist = (double)dist_;

    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,
```

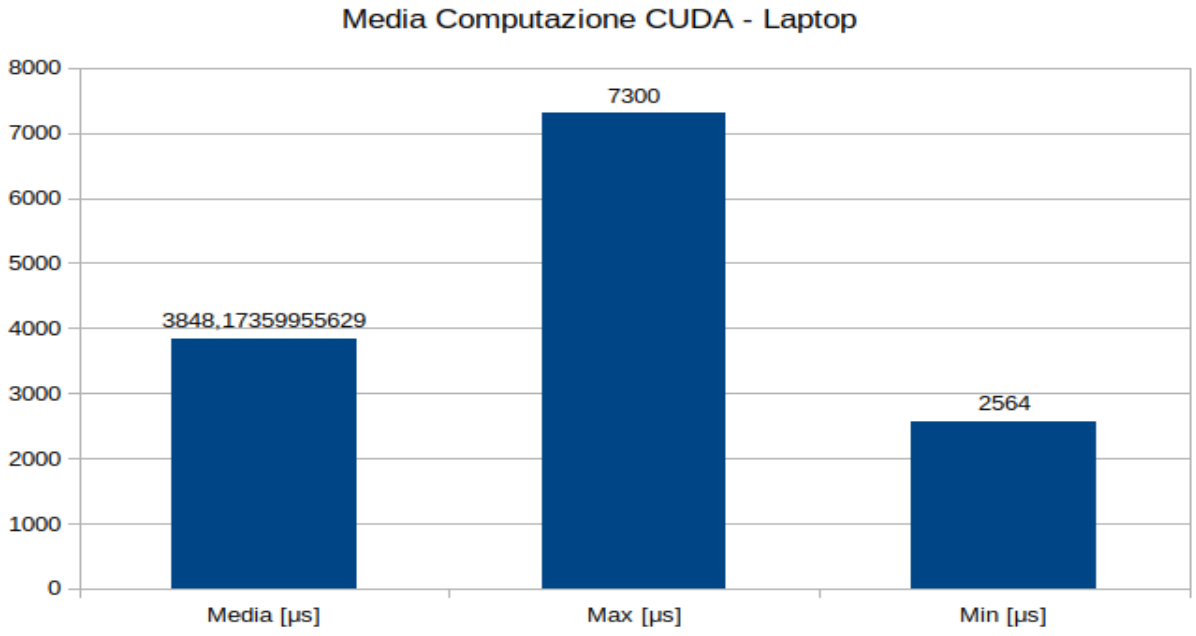
```
    sin(gamma), cos(gamma), 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1  );

// R - rotation matrix
Mat R = RX * RY * RZ;
// cout<< " R : \n "<< R << endl;


// 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));

warpPerspectiveRemappingCUDA(input, output, transformationMat);

return;
}
```

Dalle analisi sperimentali si osserva che per :

1) LAPTOP:
Il tempo medio impiegato per effettuare le moltiplicazioni tra matrici in CPU grazie all'overload messo a disposizione da OpenCV è il seguente:



il tempo medio della computazione è :



Quindi in media per un singolo frame si impiegano 3848 µs, calcolando lo SpeedUp come :

$$SpeedUp = \frac{MediaTempiOpenCV[\mu s]}{MediaTempiCUDA[\mu s]}$$

e ricordando che sul laptop si aveva un *tempo medio di OpenCv* pari a 8854 µs, si ottiene che

SpeedUP= 8854 ÷ 3848 = 2,30

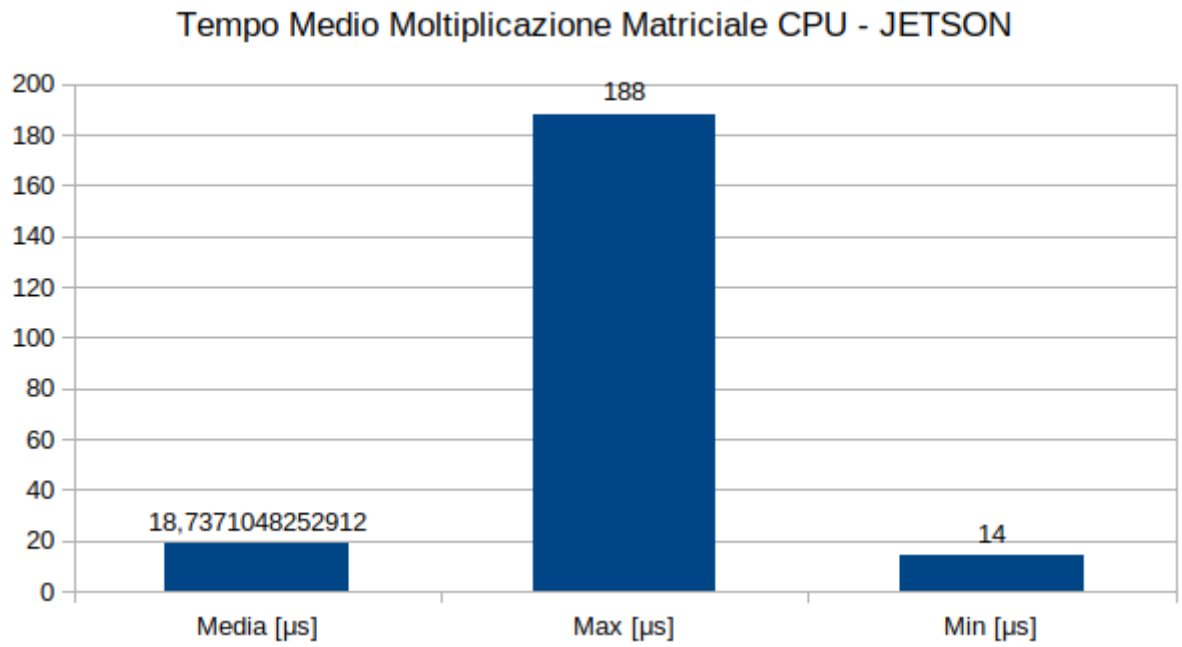
Allo stesso modo, calcolando, lo SpeedUp tra lo stesso algoritmo che gira solamente sul CPU che impiega in media 29438 µs e quello "ibrido" appena esposto si ottiene

$$SpeedUp = \frac{tempiCPU}{tempiGPU}$$

SpeedUp = 29438 ÷ 3848 = 7,65

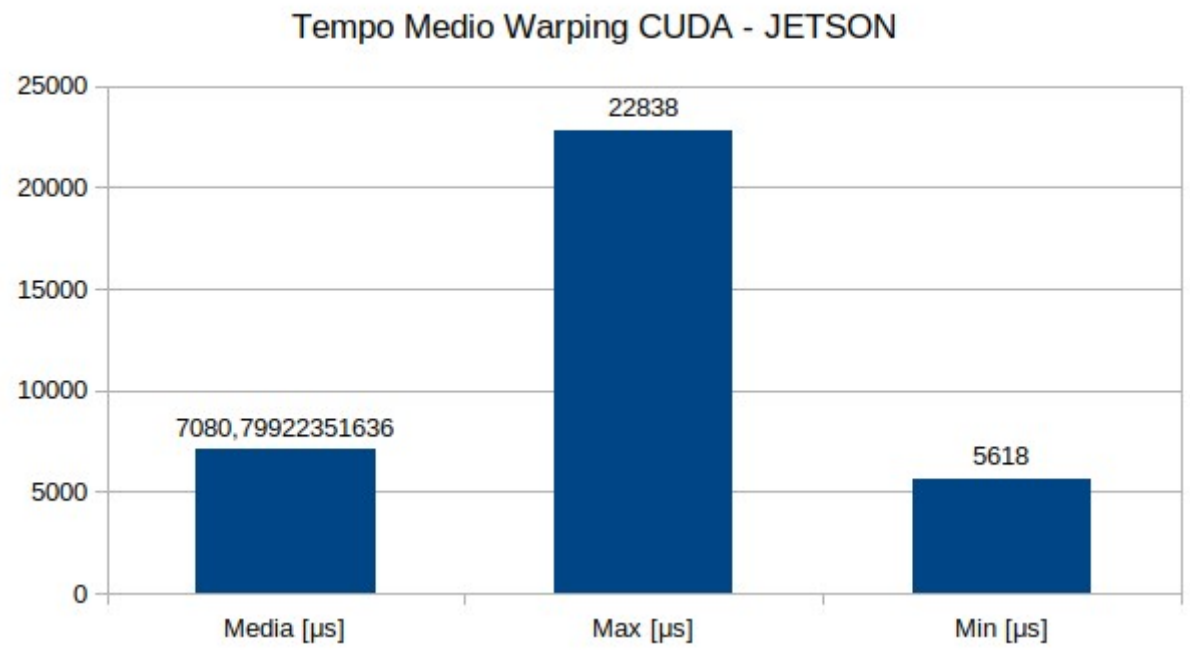
1) JETSON NANO:

Anche nel jetson si osserva che il tempo medio per effettuare le moltiplicazioni matriciali per la traslazione dei pixel è molto basso, infatti ne risulta il seguente grafico:



Da questo grafico si può capire quindi che anche su tale piattaforma embedded non si ha un guadagno di performance effettuando l'offload della moltiplicazione su un kernel CUDA, anzi si peggiorano le prestazioni.

Perciò relegando tale calcolo al CPU ed effettuando la trasposizione in CUDA si ottiene ilseguente grafico:



Quindi, ricordando che per analizzare un frame con l'algoritmo richiamante sole API di OpenCV si impiega in media 16226 µs, calcolando lo SpeedUp come riportato si ottiene :

$$SpeedUp = \frac{MediaTempiOpenCV[\mu s]}{MediaTempiCUDA[\mu s]}$$

SpeedUP= 16226 ÷ 7080 = 2,29

Allo stesso modo, calcolando, lo SpeedUp tra lo stesso algoritmo che gira solamente sul CPU che impiega in media 79266 µs e quello "ibrido" appena esposto si ottiene

$$SpeedUp = \frac{tempiCPU}{tempiGPU}$$

SpeedUp = 79266 ÷ 7080 = 11,19