

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

Sommario

Introduzione.....	3
a. Finalità del progetto.....	3
b. Stato dell'arte.....	3
Descrizione.....	5
a. Schema a Blocchi.....	5
b. Guida per L'utente.....	6
c. Codice Sorgente e Funzioni Accelerabili.....	7
Conclusioni.....	19
a. Benchmarking e conclusioni finali.....	19
b. Benchmarking su Nvidia Jetson Nano®.....	22

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'angolazione differente da quella realmente ripresa. Tale trasposizione deve essere effettuata nel più breve 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

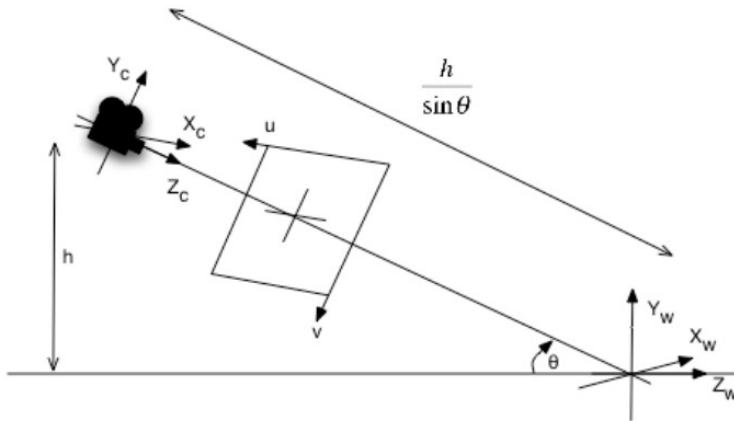
Infine, si è eseguito tale programma, su diverse architetture e si è calcolato lo speed up

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

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -h/\sin\theta \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$K = \begin{bmatrix} f * ku & s & u_0 & 0 \\ 0 & f * kv & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

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 ($p_{11} \dots p_{34}$) è il risultato della moltiplicazione $K T R$ essendo interessati al piano della strada ($Y_w = 0$), la precedente formula diventa:

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

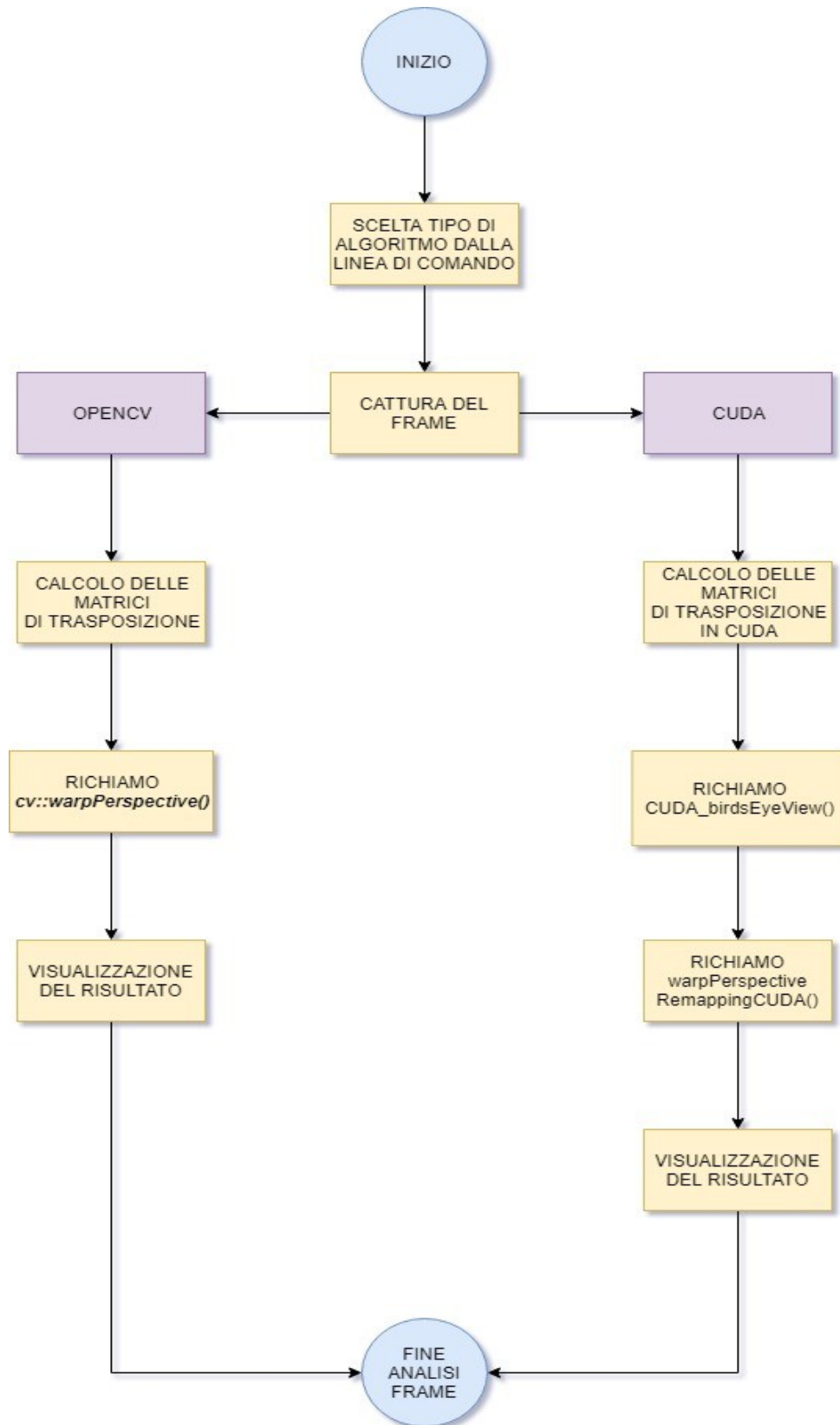
Grazie a quest'ultima equazione è possibile ottenere il mapping top down desiderato.

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/ ' | 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();
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 | WARP_INVER
SE_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] = {
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*, 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 sì, 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 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.

```

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:

$$A[0] = 2 \quad * \quad B[0] = 6$$

$$A[1] = 3 \quad * \quad B[2] = 8$$

$$C[0] = 36$$

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

$$A[0] = 2 \quad * \quad B[1] = 7$$

$$A[1] = 3 \quad * \quad B[3] = 9$$

$$C[1] = 41$$

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 numRows, int numColumns,
14.   int numBRows, int numBColumns){
15.     dim3 blockDim(16, 16);
16.     dim3 gridDim(ceil(((float)numColumns) / blockDim.x), ceil(((float)numBRows) / blockDim
17.       .y));
18.     float *d_A, *d_B, *d_C;
19.     cudaStatus = cudaMalloc((void **) &d_A, sizeof(float)*numRows*numColumns);
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)*numRows * 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)*numRows*numColumns, cudaMemcpyHostToDevice);
36.     cudaMemcpy(d_B, B, sizeof(float)*numBRows*numBColumns, cudaMemcpyHostToDevice);
37.     cudaMemset(d_C, 0, numRows * numBColumns * sizeof(float));
38.
```

```

39.     generic_mat_mul<<<gridDim, blockDim>>>(d_A, d_B, d_C, numRows, numAColumns, numBRows,
numBColumns);
40.     cudaThreadSynchronize();
41.     cudaStatus = cudaGetLastError();
42.     if (cudaStatus != cudaSuccess) {
43.         fprintf(stderr, "Kernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
44.         goto Error;
45.     }
46.
47.     cudaStatus = cudaDeviceSynchronize();
48.     if (cudaStatus != cudaSuccess) {
49.         fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addK
ernel!\n", cudaStatus);
50.         goto Error;
51.     }
52.
53.     cudaStatus = cudaMemcpy(C, d_C, numRows * numBColumns * sizeof(float), cudaMemcpyDevic
eToHost);
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 “*calculateTransferArray()*” si vanno a calcolare le nuove posizioni dei pixel sulla base della matrice H , che è il risultato dei precedenti prodotti matriciali ($K * T * R$)

```

1.  cudaError_t warpPerspectiveRemappingCUDA(Mat input, Mat &output, const Mat H){
2.      cudaError_t cudaStatus;
3.      // allocate array of all locations
4.      int Numrows = input.rows;
5.      int Numcols = input.cols;
6.      int channels  = input.channels();
7.      int size = Numrows*Numcols;
8.
9.      int *TransArray = (int *)malloc(sizeof(int)*size);
10.
11.     cudaStatus = calculateTransferArray(H,TransArray,Numrows, Numcols);
12.
13.     cout <<" \n richiamo la funzione per il remapping \n";
14.     cout <<" \n NUMERO DI CANALI : " << input.channels() << "\n";
15.
16.     output = remappingMultiChannelImage(input, TransArray);
17.
18.     return cudaStatus;
19. }

```

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.    }
}
```

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:

R	G	B	R	G	B
R	G	B	R	G	B

ciascun thread prende la tripletta *RGB*, e la trascrive nella nuova posizione.

Il calcolo del vettore di traslazione avviene in questo modo:

```
1. __global__ void calc_tranf_array(float *H, int *transfArray, int numRows, int numColumns) {
2.     //int row = blockIdx.y * blockDim.y + threadIdx.y;
3.     int idx = blockIdx.x * blockDim.x + threadIdx.x;
4.     int MaxX,MaxY = -1000;
5.     int MinX,MinY = 1000;
6.     int homeX, homeY;
7.     if (idx < numRows * numColumns) {
8.         homeX=idx % numColumns;
9.         homeY=idx / numColumns;
10.
11.         float x = (H[0] * (homeX)) +( H[1] * (homeY)) + H[2] ;
12.         float y = (H[3] * (homeX)) +( H[4] * (homeY)) + H[5] ;
13.         float s = (H[6] * (homeX)) +( H[7] * (homeY)) + H[8] ;
14.         x = floor(x/s);
15.         y = floor(y/s);
16.
17.         if (homeX == 0){
18.             if (x > MaxX) MaxX = x;
19.             if (x < MinX) MinX = x;
20.         }
21.
22.         if (homeY == 0){
23.             if (y > MaxY) MaxY = y;
24.             if (y < MinY) MinY = y;
25.         }
26.         if( y >= numRows || y<0 || x >= numColumns || x < 0){
27.             transfArray[idx] = -1;
28.         }else{
29.             transfArray[idx] = (y * numColumns + x);
30.         }
31.     }
32. }
```

Tale funzione risolve il problema della ricerca di una traslazione omografica bidimensionale.

Si hanno tanti thread quanti sono i pixel dell'immagine di partenza, in questo modo si ha un mapping uno ad uno, terminata la computazione si va a scrivere la posizione sul vettore

L'omografia bidimensionale consiste nella determinazione di una trasformazione in grado di mappare punti di un piano in punti di un altro piano.

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 $h_{11} \dots h_{33}$ 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:

$$\begin{aligned} x'_{i1} &= \frac{h_{1,0} + h_{1,1}x_{i1} + \dots + h_{1,n}x_{in}}{h_{0,0} + h_{0,1}x_{i1} + \dots + h_{0,n}x_{in}} \\ &\vdots \\ x'_{in} &= \frac{h_{n,0} + h_{n,1}x_{i1} + \dots + h_{n,n}x_{in}}{h_{0,0} + h_{0,1}x_{i1} + \dots + h_{0,n}x_{in}} \end{aligned}$$

Di seguito, viene riportato il wrapper che calcola tale vettore di traslazione, che avrà tante posizioni, quanti sono i pixel dell'immagine da traslare, e che in seguito richiama la funzione che effettua realmente tale trasposizione.

```

22. cv::Mat remappingMultiChannelImage(Mat image, int *tranfArray){
23.     cudaError_t cudaStatus;
24.     dim3 blockDim(16, 16);
25.     dim3 gridDim(ceil((float)image.cols / blockDim.x), ceil((float)image.cols / blockDim.y), 1);
26.     int num_RGBelem,size = image.rows * image.cols;
27.     Mat null_mat = Mat::zeros(cv::Size(image.rows, image.cols), CV_8UC3);
28.     cout << "Remapping image :\n \t \t tipo matrice : " << "CV_" + type2str(image.type()) << endl;
29.     cv::Mat img;
30.     uchar *d_image, *d_output;
31.     int *d_tranfArray;
32.     //definisco l'immagine
33.     cv::cuda::GpuMat input, output;
34.     input.upload(image);
35.     output = input.clone();
36.     output.setTo(Scalar::all(0));
37.     //cout << " \n alloco il vettore di transposizione \n";
38.     cudaStatus = cudaMalloc((void **) &d_tranfArray, sizeof(int) * size);
39.     if (cudaStatus != cudaSuccess) {
40.         fprintf(stderr, "cudaMalloc failed!");
41.         goto ErrorNewMultiRemapping;
42.     }
43.
44.     //cout << " \n copio il vettore di transposizione \n";
45.     cudaStatus = cudaMemcpy(d_tranfArray,tranfArray,sizeof(int) * size, cudaMemcpyHostToDevice);
46.     if (cudaStatus != cudaSuccess) {
47.         fprintf(stderr, "CudaMemSetfailed: %s\n", cudaGetErrorString(cudaStatus));
48.         goto ErrorNewMultiRemapping;
49.     }
50.
51.     new_remapping_kernel<<<gridDim,blockDim>>> (input, input.rows, input.cols, input.step, image.channels(), d_tranfArray, output);
52.     cudaThreadSynchronize();
53.     output.download(img);
54.     return img;
55.
56. ErrorNewMultiRemapping:
57.     cudaFree(d_tranfArray);
58.     exit(0);
59. }

```

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. Benchmarking e conclusioni finali

L'algoritmo sopra presentato, è stato eseguito in tre diversi dispositivi, aventi capacità 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 channel
- 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.

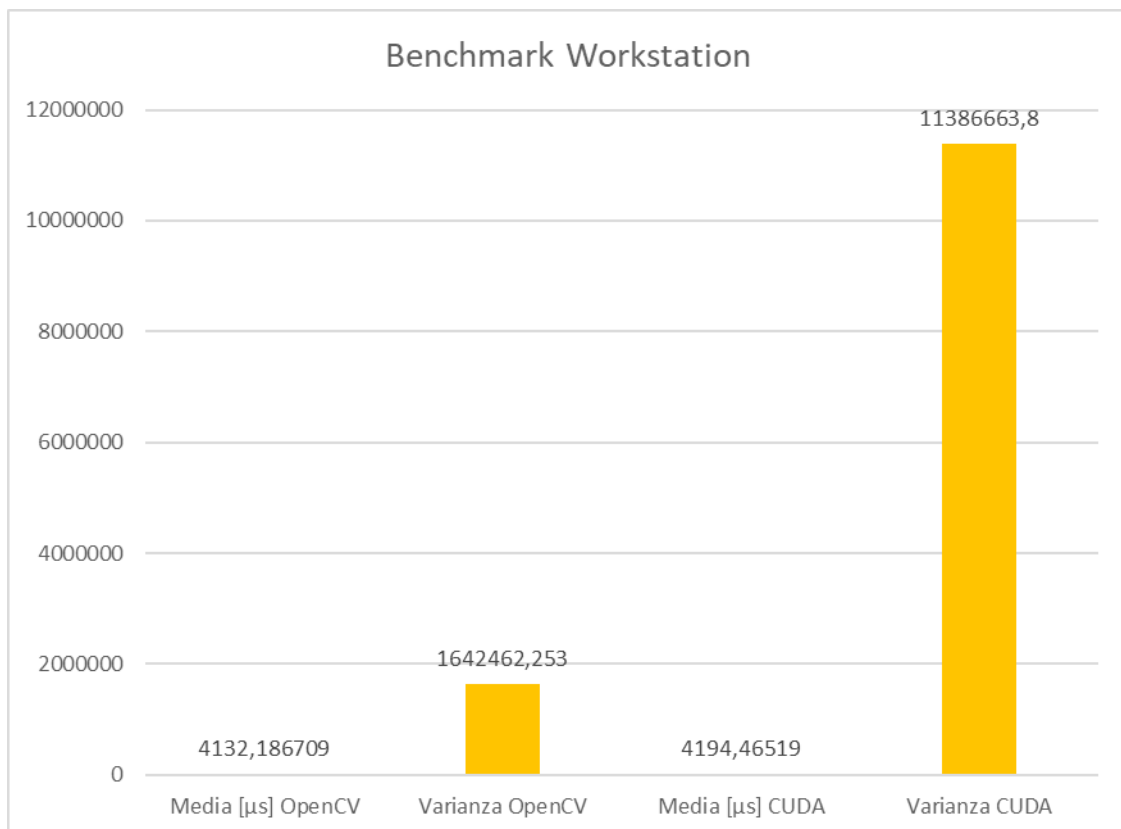
Per avere una più ampia conoscenza delle varie tempistiche che le varie parti del codice CUDA impiegano ad essere eseguite, si è scelto di misurare il tempo delle *cudaMalloc* e *cudaMemcpy*.

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.

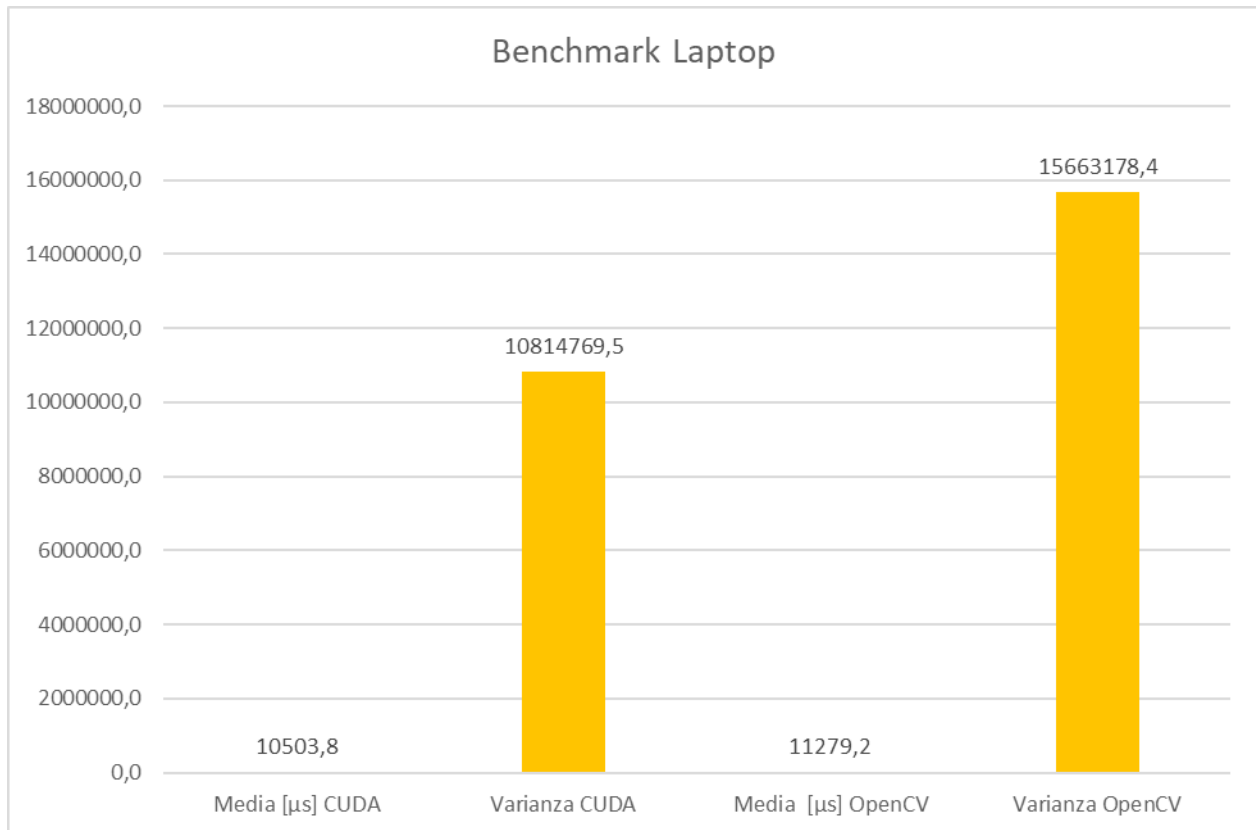
Ottenendo uno $SpeedUP = 0,98515222$

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

Analizzando, invece, sempre lo stesso stream video, con gli stessi parametri, ma con un diverso hardware:

si ottengono i seguenti risultati:



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

$$SpeedUP = 1,073820$$

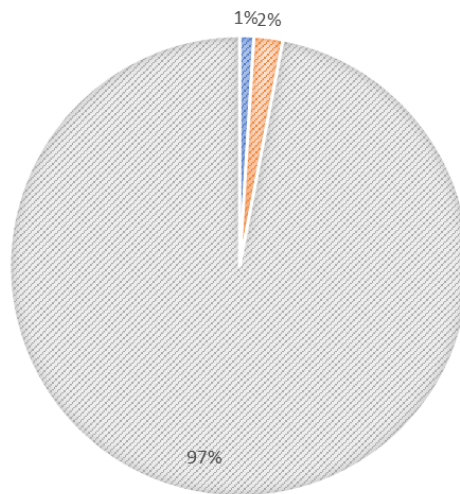
Da ciò deriva che la versione parallelizzata risulta essere di poco migliore rispetto a quella utilizzando solo direttive OpenCV.

Come esposto prima, nel codice CUDA, vengono effettuate delle operazioni, come *cudamalloc* e *cudamemcpy* ad ogni richiamo di funzione con annessa *free*. Esse richiedono un elevato sforzo computazionale che potrebbe essere eliminato. Ai fini progettuali, si è scelto di non intervenire sul codice ma di capire il loro impatto sul tempo medio totale.

Si ottiene così il seguente grafico a torta:

BENCHMARKING LAPTOP EXECUTION TIME CUDA

■ Media [μs] memcpy matmul ■ Media [μs] memcpy remapping ■ Parte restante computazione [μs]

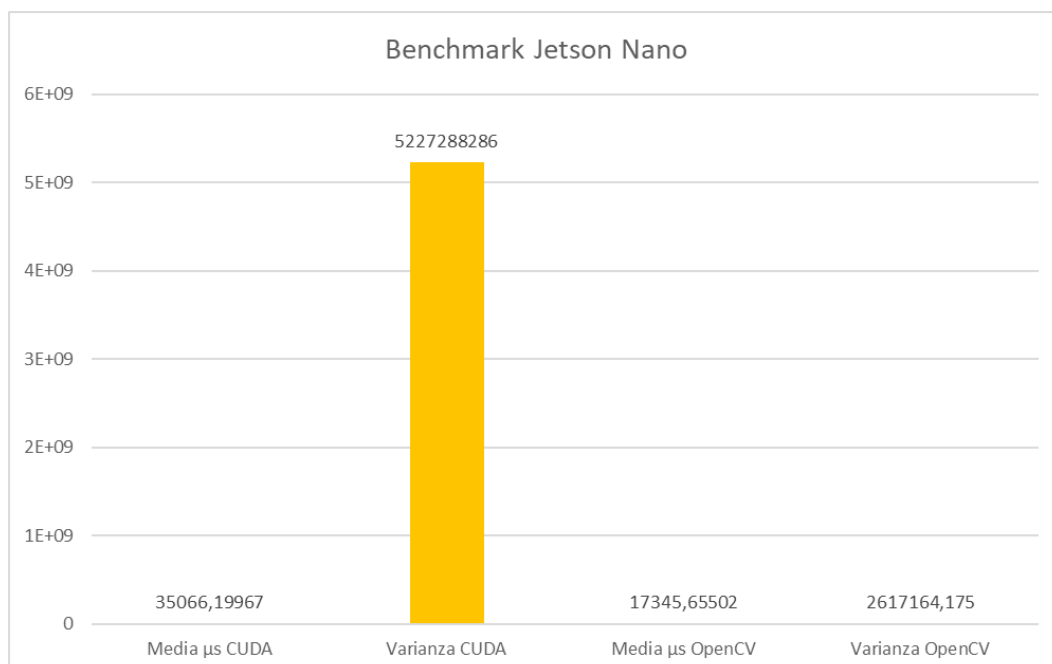


Dal grafico si può notare che solo il 3%, sul totale del tempo di esecuzione, è necessario per portare a termine sia le *cudaMalloc* che le *cudaMemcpy*. Tale misura consiste nell'ipotetico guadagno in termini di performance, dell'algoritmo CUDA.

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, la memoria ram è condivisa sia dal processore che dalla GPU, quindi non si ha più il collo di bottiglia introdotto dal bus PCI.

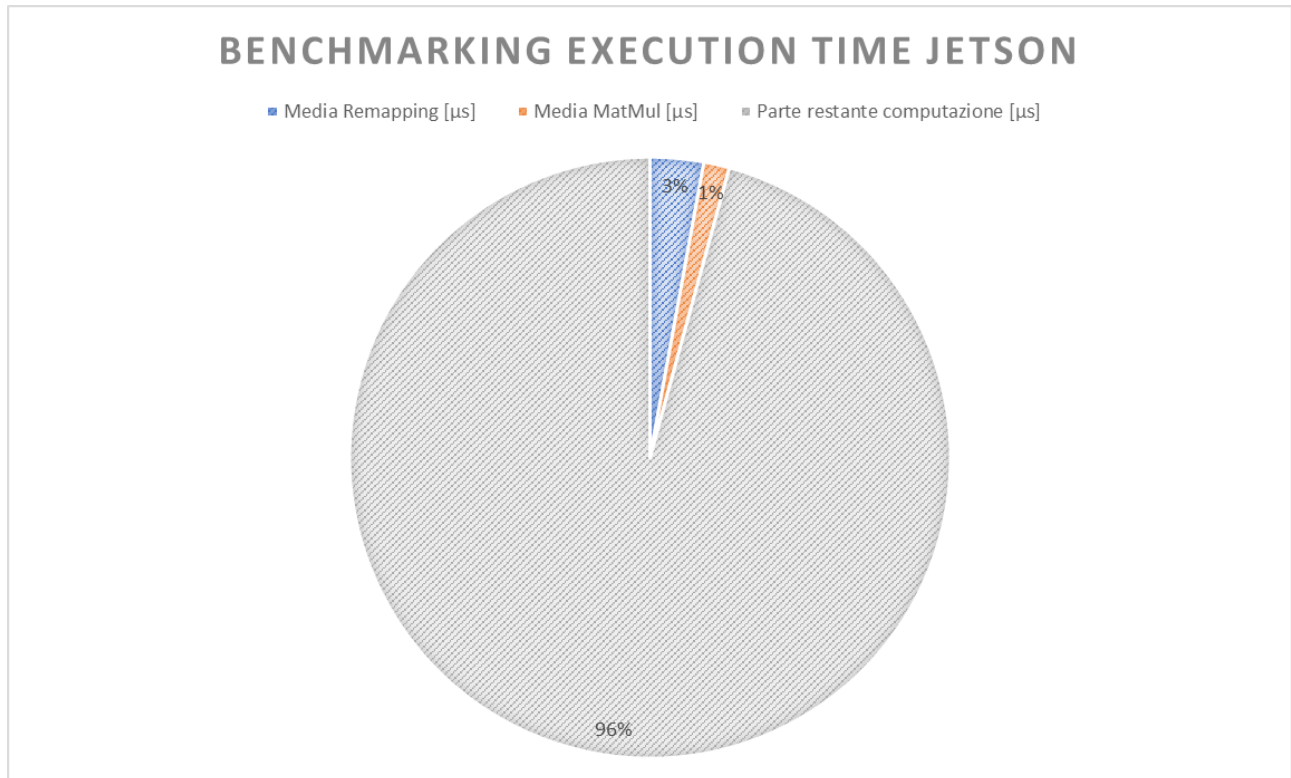


Misurando lo speedUP si ottiene:

$$SpeedUP = 0,494654544$$

Questo risultato porta ad osservare che, l'algoritmo che sfrutta i kernel CUDA peggiora le performace di circa il 50%, rispetto alla versione OpenCV.

Come per il Laptop, è interessante capire invece come incidono sull'algoritmo le varie parti funzionali del codice, ciò lo si può capire da seguente grafico a torta:



Dal grafico si può evincere un risultato molto interessante, ovverosia, che le tempistiche delle computazioni dei prodotti matriciali e del calcolo del vettore di trasposizione sono assimilabili al caso precedente, ovvero quello del laptop, ma dal grafico dei tempi, possiamo capire che la computazione è molto più lenta, infatti in media si impiegano 35000 μs a frame, anziché i circa 10000 μs.

Per aver una concezione migliore della differenza di performance, calcolando il seguente rapporto :

$$SpeedUp = \frac{MediaTempiLaptopCUDA}{MediaTempiJetsonCUDA}$$

si ottiene un rapporto di 0,2995 che espresso in percentuale, si può capire di avere quasi una perdita di performance del 70%