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 3](#__RefHeading___Toc1208_2846315957)

[a. Finalità del progetto 3](#__RefHeading___Toc1210_2846315957)

[b. Stato dell’arte 3](#__RefHeading___Toc211_3904458324)

[Descrizione 5](#__RefHeading___Toc213_3904458324)

[a. Schema a Blocchi 5](#__RefHeading___Toc1216_2846315957)

[b. Guida per L’utente 6](#__RefHeading___Toc1216_28463159571)

[c. Codice Sorgente e Funzioni Accelerabili 7](#__RefHeading___Toc2115_1009112065)

[Conclusioni 18](#__RefHeading___Toc2117_1009112065)

[a. Benchmarking e conclusioni finali 18](#__RefHeading___Toc1216_28463159572)

[b. Benchmarking su Nvidia Jetson Nano® 20](#__RefHeading___Toc1559_1593546379)

# Introduzione

# Finalità del progetto

In questa relazione viene descritto il procedimento per effettuare la trasformazione “Bird's Eye View”. Tale trasformazione lineare 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.

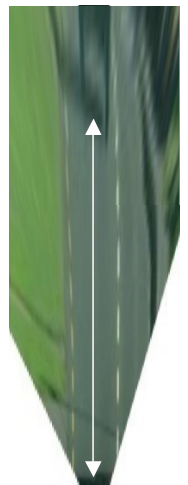
L’utente ha la possibilità di effettuare la trasposizione dei pixel tramite:

* Funzioni già esistenti di OpenCV
* Funzioni accelerate in CUDA

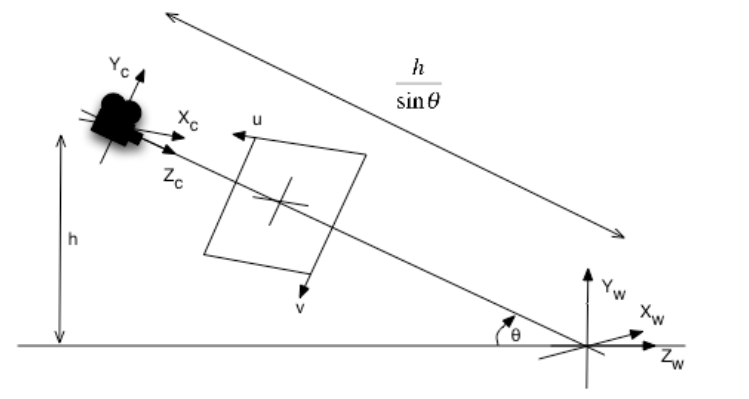
Tale trasposizione deve essere effettuata nel più beve tempo possibile in modo da non creare troppo ritardo, ed un calo degli FPS. Dopo una introduzione pratica del lavoro, verranno esposti i risultati sperimentali.

# 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. 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 = T = K =

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 :

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

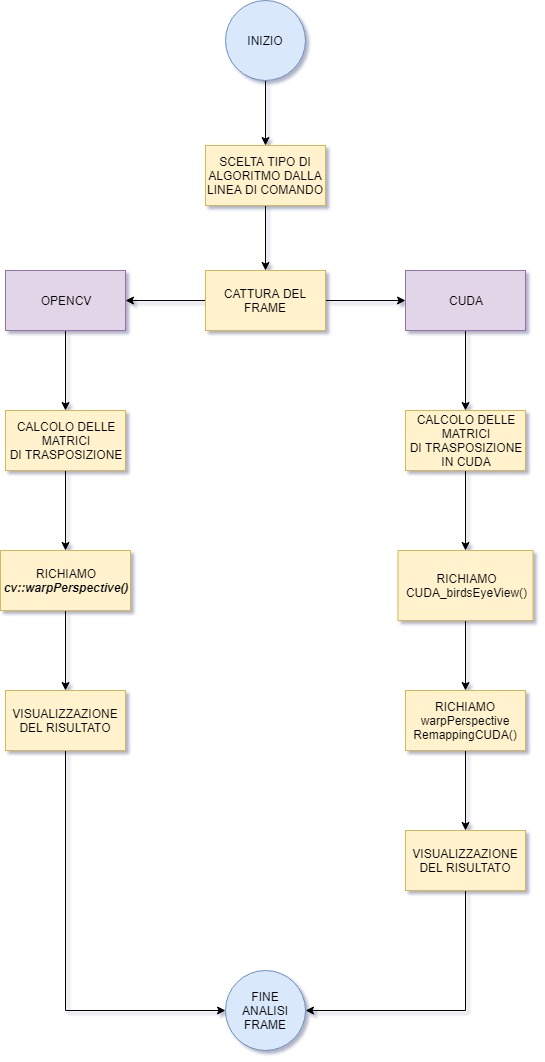
Grazie a quest’ultima equazione è possibile ottenere il mapping top down desiderato.

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

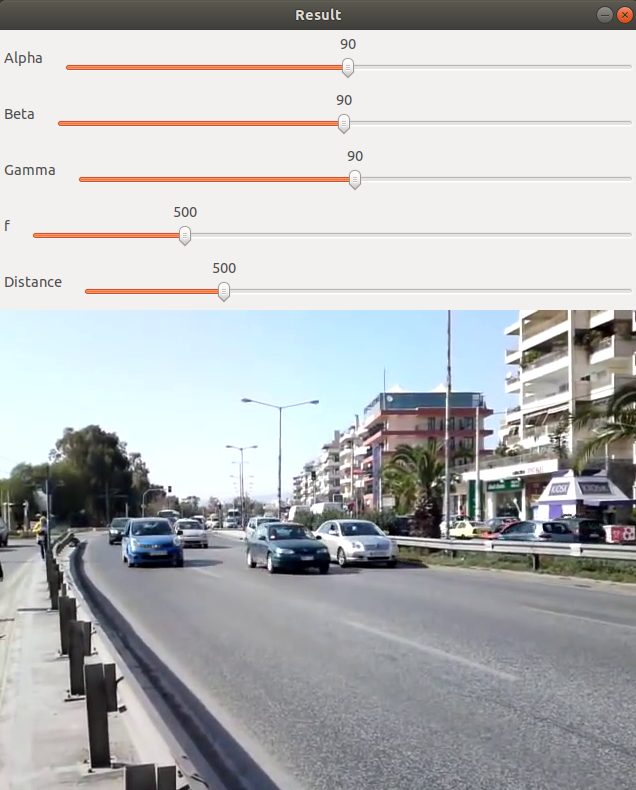
# Schema a Blocchi



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.

# 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>

# 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[]) {
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;
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";
20. }
22. **if** (argc == 2){
23. capture.open(0);
24. }
25. **if** (argc == 3){
26. string filename = argv[2];
27. capture.open(filename);
28. }
30. **if**(!capture.isOpened()) **throw** "Error reading video";
32. namedWindow("Result", 1);
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);
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));
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::microseconds>(end - begin).count() << "[µs]" << std::endl;
57. std::cout << "Time difference = " << std::chrono::duration\_cast<std::chrono::nanoseconds> (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::microseconds>(end - begin).count() << "[µs]" << std::endl;
63. std::cout << "Time difference = " << std::chrono::duration\_cast<std::chrono::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. }

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;
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\_;
10. Size input\_size = input.size();
11. **double** w = (**double**)input\_size.width, h = (**double**)input\_size.height;

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 );
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 );
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  );
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\_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){
3. cudaError\_t error;
5. **double** focalLength, dist, alpha, beta, gamma;
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\_;
13. Size input\_size = input.size();
14. **double** w = (**double**)input\_size.width, h = (**double**)input\_size.height;

17. **float** A1[12] = {
18. 1, 0, -w/2,
19. 0, 1, -h/2,
20. 0, 0, 0,
21. 0, 0, 1
22. };
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. };
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. };
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. };
45. // cout << "stampo RX \n";
46. // stampaMatrice(RX , 4, 4);
47. // R - rotation matrix
48. // Mat R = RX \* RY \* RZ;
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. }
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. };
76. //Mat transformationMat = K \* (T \* (R \* A1));
77. **float** R\_A1[12], T\_RA1[12], transformationvector[9];
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);
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);
95. error = matrixMultiplication(K, T\_RA1, transformationvector, 4, 4, 4, 3);
96. **if** (error != cudaSuccess) {
97. fprintf(stderr, "cudaMalloc failed!");
98. exit(0);
99. }
101. cv::Mat tranf\_mat(3,3,CV\_32FC1);
103. arrayToMat(tranf\_mat,transformationvector,9);
105. warpPerspectiveRemappingCUDA(input, output, tranf\_mat);
107. **return**;
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** 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. }

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.

1. cudaError\_t matrixMultiplication(**float** \*A, **float** \*B, **float** \*C, **int** numARows,**int** numAColumns, **int** numBRows, **int** numBColumns){
3. dim3 blockDim(16, 16);
4. dim3 gridDim(ceil(((**float**)numAColumns) / blockDim.x),ceil(((**float**)numBRows) / blockDim.y));
6. **float** \*d\_A, \*d\_B, \*d\_C;
7. cudaStatus = cudaMalloc((**void** \*\*) &d\_A, **sizeof**(**float**)\*numARows\*numAColumns);
8. **if** (cudaStatus != cudaSuccess) {
9. fprintf(stderr, "cudaMalloc failed!");
10. **goto** Error;
11. }
12. cudaStatus = cudaMalloc((**void** \*\*) &d\_B, **sizeof**(**float**)\*numBRows\*numBColumns);
13. **if** (cudaStatus != cudaSuccess) {
14. fprintf(stderr, "cudaMalloc failed!");
15. **goto** Error;
16. }
17. cudaStatus = cudaMalloc((**void** \*\*) &d\_C, **sizeof**(**float**)\*numARows \* numBColumns);
18. **if** (cudaStatus != cudaSuccess) {
19. fprintf(stderr, "cudaMalloc failed!");
20. **goto** Error;
21. }
22. //copio i vettori
23. cudaMemcpy(d\_A,A,**sizeof**(**float**)\*numARows\*numAColumns,cudaMemcpyHostToDevice);
24. cudaMemcpy(d\_B,B,**sizeof**(**float**)\*numBRows\*numBColumns,cudaMemcpyHostToDevice);
25. cudaMemset(d\_C, 0, numARows \* numBColumns \* **sizeof**(**float**));
27. generic\_mat\_mul<<<gridDim, blockDim>>>(d\_A, d\_B, d\_C, numARows, numAColumns, numBRows, numBColumns);
28. cudaThreadSynchronize();
29. cudaStatus = cudaGetLastError();
30. **if** (cudaStatus != cudaSuccess) {
31. fprintf(stderr, "Kernel launch failed: %s\n", cudaGetErrorString(cudaStatus));
32. **goto** Error;
33. }
35. cudaStatus = cudaDeviceSynchronize();
36. **if** (cudaStatus != cudaSuccess) {
37. fprintf(stderr, "cudaDeviceSynchronize returned error code %d after launching addKernel!\n", cudaStatus);
38. **goto** Error;
39. }
41. cudaStatus = cudaMemcpy(C, d\_C,numARows \* numBColumns \* **sizeof**(**float**), cudaMemcpyDeviceToHost);
42. **if** (cudaStatus != cudaSuccess) {
43. fprintf(stderr, "cudaMemcpy failed!");
44. **goto** Error;
45. }
46. //@@ Free the GPU memory here
47. Error:
48. cudaFree(d\_A);
49. cudaFree(d\_B);
50. cudaFree(d\_C);
51. **return** cudaStatus;
53. }

***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;
4. // allocate array of all locations
5. **int** Numrows = input.rows;
6. **int** Numcols = input.cols;
7. **int** channels   = input.channels();
8. **int** size = Numrows\*Numcols;
10. **int** \*TransArry = (**int** \*)malloc(**sizeof**(**int**)\*size);
12. cudaStatus = calculateTransferArray(H,TransArry,Numrows, Numcols);
13. cout <<" \n richiamo la funzione per il remapping \n";
14. cout <<" \n NUMERO DI CANALI : " << input.channels() << "\n";
16. output = remappingMultiChannelImage(input, TransArry);
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.

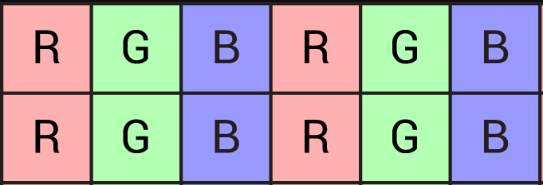
Ovviamente è possibile accedere a tale zona tramite a dei puntatori, definiti dalla libreria stessa.

1. \_\_global\_\_ **void** new\_remapping\_kernel(cv::cuda::PtrStepSz<uchar3> src, **int** numRows, **int** numCols, **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 ipixel dell’immagine.

L’immagine RGB analizzata ha i canali interallacciati, ciò significa che la matrice è così fatta:

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

Il calcolo del vettore di traslazione avviene in questo modo:

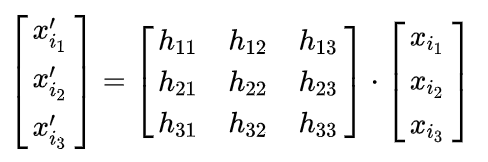
1. \_\_global\_\_ **void** calc\_tranf\_array(**float** \*H, **int** \*transfArray, **int** numARows, **int** numAColumns) {
2. //int row = blockIdx.y \* blockDim.y + threadIdx.y;
3. **int** idx = blockIdx.x \* blockDim.x + threadIdx.x;
4. **int** MaxX,MaxY = -1000;
5. **int** MinX,MinY =  1000;
6. **int** homeX, homeY;
7. **if** (idx < numARows \* numAColumns) {
8. homeX=idx % numAColumns;
9. homeY=idx / numAColumns;
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);
17. **if** (homeX == 0){
18. **if** (x > MaxX) MaxX = x;
19. **if** (x < MinX) MinX = x;
20. }
22. **if** (homeY == 0){
23. **if** (y > MaxY) MaxY = y;
24. **if** (y < MinY) MinY = y;
25. }
26. **if**( y >= numARows || y<0 || x >= numAColumns ||  x < 0){
27. transfArray[idx]  = -1;
28. }**else**{
29. transfArray[idx] = (y \* numAColumns + 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

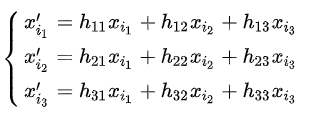
L'omografia bidimensionale consiste nella determinazione di una trasformazione in grado di mappare punti di un piano in punti di un altro piano.

Per trovare tale mapping si applica la seguente formula



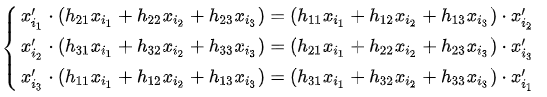
dove gli *X’i* sono le nuove coordinate, mentre gli *Xi,* sono le vecchie posizioni.

Sviluppando la precedente formula si ha :

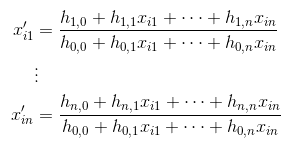


Risolvendo tale sistema si ottiene solamente 3 equazioni non dipendenti. Infatti i parametri *h11 .. h33* non devono essere considerati singolarmente, ma va considerato il loro rapporto tra essi, ottenendo così solamente 8 gradi di libertà.

Ottenendo:



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



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.

1. cv::Mat remappingMultiChannelImage(Mat image, **int** \*tranfArray){
2. cudaError\_t cudaStatus;
3. dim3 blockDim(16, 16);
4. dim3 gridDim(ceil((**float**)image.cols / blockDim.x), ceil((**float**)image.cols / blockDim.y), 1);
5. **int** num\_RGBelem,size = image.rows \* image.cols;
6. Mat null\_mat = Mat::zeros(cv::Size(image.rows, image.cols), CV\_8UC3);
7. cout << "Remapping image :\n \t \ttipo matrice :" << "CV\_" + type2str(image.type()) <<endl;
8. cv::Mat img;
9. uchar \*d\_image, \*d\_output;
10. **int** \*d\_tranfArray;
11. //definisco l'immagine
12. cv::cuda::GpuMat input, output;
13. input.upload(image);
14. output = input.clone();
15. output.setTo(Scalar::all(0));
16. //cout <<" \n alloco il vettore di transposizione \n";
17. cudaStatus = cudaMalloc((**void** \*\*) &d\_tranfArray, **sizeof**(**int**) \* size);
18. **if** (cudaStatus != cudaSuccess) {
19. fprintf(stderr, "cudaMalloc failed!");
20. **goto** ErrorNewMultiRemapping;
21. }
23. //cout <<" \n copio il vettore di transposizione \n";
24. cudaStatus = cudaMemcpy(d\_tranfArray,tranfArray,**sizeof**(**int**) \* size, cudaMemcpyHostToDevice);
25. **if** (cudaStatus != cudaSuccess) {
26. fprintf(stderr, "CudaMemSetfailed: %s\n", cudaGetErrorString(cudaStatus));
27. **goto** ErrorNewMultiRemapping;
28. }
30. new\_remapping\_kernel<<<gridDim,blockDim>>> (input, input.rows, input.cols, input.step, image.channels(), d\_tranfArray, output);
31. cudaThreadSynchronize();
32. output.download(img);
33. **return** img;
35. ErrorNewMultiRemapping:
36. cudaFree(d\_tranfArray);
37. exit(0);
38. }

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 capacita computazionali diverse.

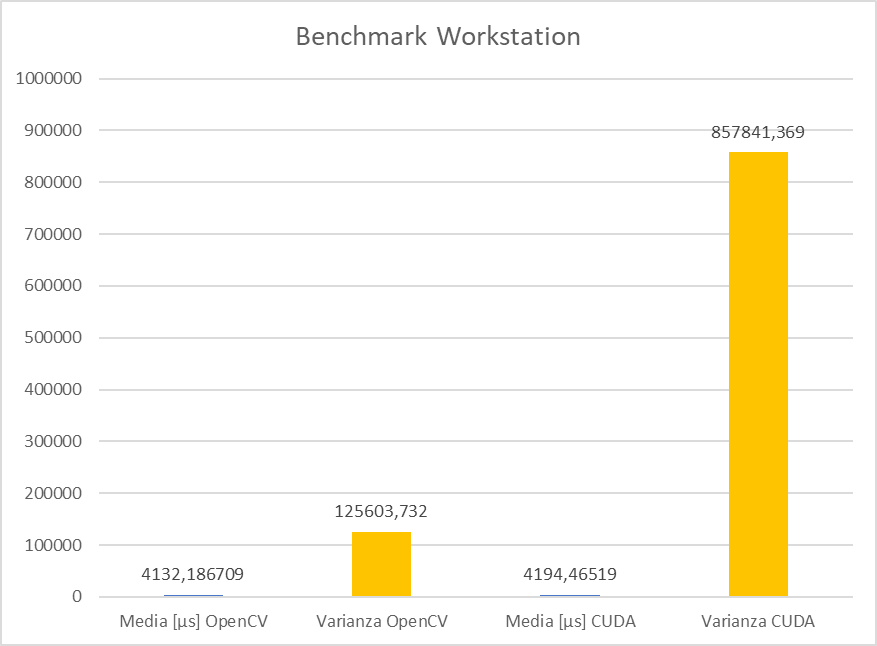
I benchmark riportati qui di seguito fanno riferimento all’analisi di un video, allegato al progetto, con parametri di riproiezione fissati, in modo da non ottenere una forte distorsione, poiché ciò comporterebbe un’elevata quantità di interpolazione dei pixel.

Il primo benchmark riportato è stato effettuato con il seguente hardware :

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

*GeForce GTX 750 2GB Ti/PCIe/SSE2*

*16 GB Ram 3200Mhz dual channel*



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.

Calcolando lo SpeedUp come rapporto tra le medie in μs tra la versione OpenCV e la versione CUDA si ottiene che :

*SpeedUP = 1,015071555*

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

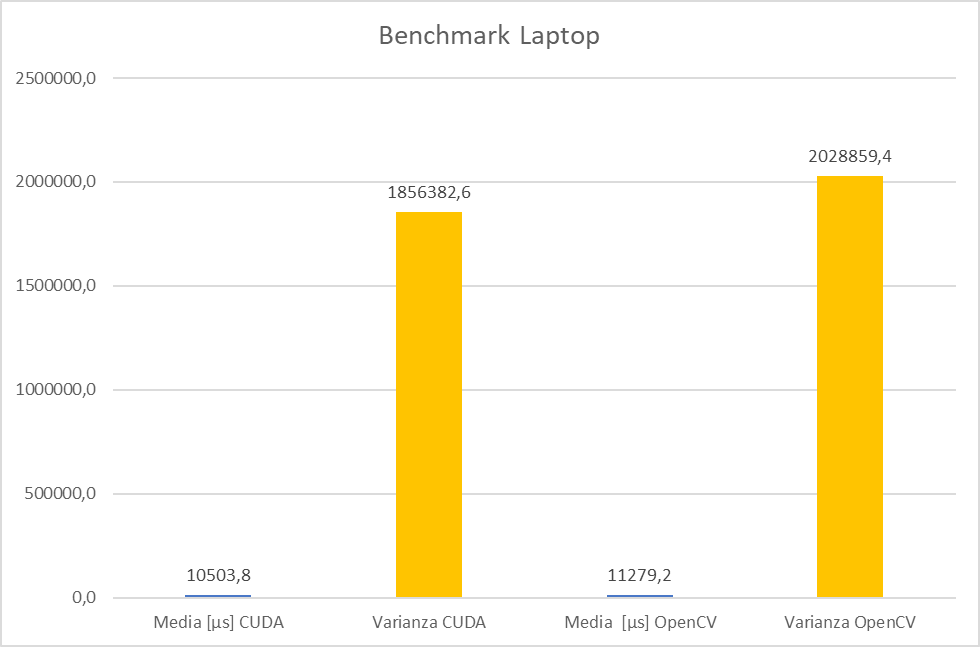
Analizzando, invece, sempre lo stesso stream video, con gli stessi parametri, ma con un diverso hardware:

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

*GeForce 920M 2GB /PCIe/SSE2*

*8 GB Ram 1600Mhz dual channell*

si ottengono i seguenti risultati:



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

*SpeedUP = 0,985152224*

Da ciò deriva che la versione parallelizzata risulta essere di poco peggiore rispetto a quella utilizzante solo direttive cuda.

# b. Benchmarking su Nvidia Jetson Nano®

In seguito sono riportati i test di esecuzione, svolti sull’architettura embedded, prodotta da Nvidia, Jetson Nano.

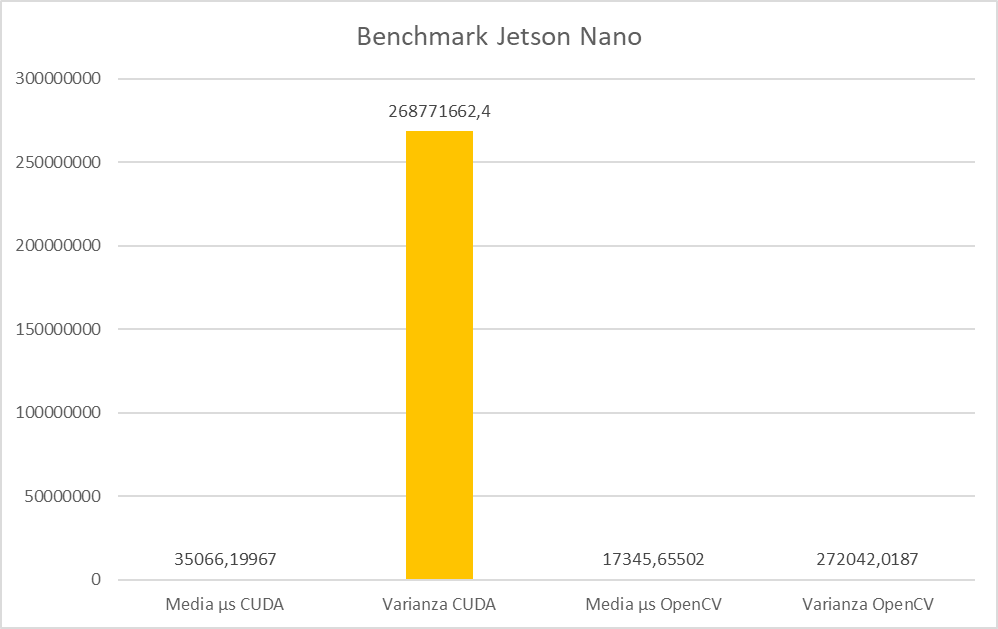
*64-bit Quad-core ARM A57 @ 1.43GHz*

*128-core NVIDIA Maxwell @ 921MHz*

*4GB 64-bit LPDDR4 @ 1600MHz | 25.6 GB/s*

In questo dispositivo, sono stati svolti i due test, sempre con lo stesso file video dei precedenti.

Dai test risulta che :



con uno speedup, misurato come sopra, ovverosia come rapporto tra media dei tempi in CUDA e media dei tempi con OpenCV, pari a:

*SpeedUP = 0,494654544*

Questo risultato porta ad osservare che, l’algoritmo che sfrutta i kernel CUDA peggiora le performace di circa il 50%

Il Seguente risultato è molto interessante, poiché il dispositivo embedded condivide lo stesso spazio di memoria sia per il CPU che per la GPU quindi non si ha più il collo di bottiglia introdotto dal BUS PCI. Il peggioramento delle performance, probabilmente, sono dovute alla gestione non ottimale della memoria della GPU, infatti l'algoritmo alloca e de alloca memoria per ogni frame da analizzare, cosa inutile, finché si ha del lavoro da svolgere.

NdR: la scheda è stata alimentata con un caricatore da smartphone con uscita nominale da 2A, è plausibile che utilizzando un’appropriata alimentazione i tempi saranno inferiori.