ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA IN

INGEGNERIA INFORMATICA

Sede di Bologna

TESI DI LAUREA

in

Calcolatori Elettronici – T

ALGORITMI PER LA CORREZIONE DI DISTORSIONI IN IMMAGINI

CANDIDATO:

Vincenzo Villani

RELATORE:

Prof. Stefano Mattoccia

Anno Accademico 2014/15

III Sessione

Indice

Capitolo 1

Il problema della distorsione

Introduzione

Per comprendere a pieno i problemi e le eventuali soluzioni proposte e discusse all'interno di questo testo, introduciamo innanzitutto il lettore ai concetti base della *Computer Vision.* Lo scopo finale di questo ramo dell'informatica, in netta ascesa negli ultimi decenni, è quello di poter ricavare informazioni sull'ambiente circostante in modo da utilizzarle nelle situazioni più disparate. Per fare ciò, si utilizza spesso una coppia di telecamere ben configurate, che, simulando il comportamento dell'occhio umano, riescono a ricostruire la scena reale in 3D, partendo da una serie di immagini, per definizione piatte, ovvero in 2D. Senza dilungarci troppo sul come sia possibile realizzare tutto ciò, possiamo pensare piuttosto che, grazie a questo meccanismo, siamo dunque in grado, per esempio, di far riconoscere ai computer gli oggetti della

scena, la loro distanza, la loro velocità, e così via. Per ottenere tutto questo però vi è tuttavia la necessità di correggere alcuni problemi. Il primo problema da risolvere è la correzione di eventuali distorsioni causate dalle lenti delle due telecamere. Le lenti infatti introducono una distorsione nelle immagini, come fanno ad esempio quelle presenti nelle *action cam*, molto di moda negli ultimi tempi, che fanno sembrare curve, linee in realtà dritte nella scena. Senza questa primissima correzione infatti, sarebbe impossibile ricostruire la scena. Il secondo problema da correggere è dovuto al mal allineamento delle due telecamere. Per i nostri scopi infatti queste dovrebbero essere sempre perfettamente allineate, e nel caso in cui non lo fossero, dovremo far sì che ci si riconduca a questa situazione ideale.

1.1 Natura e tipologia delle distorsioni

In un sistema di visione sono inevitabilmente presenti distorsioni causate dalle lenti che, in molti scenari applicati, debbono essere compensate. Inoltre, in un sistema stereo, oltre alla distorsione causata dalle ottiche è necessario applicare una trasformazione sulle immagini non distorte al fine di poter allineare virtualmente i piani immagine delle due telecamere e ottenere un sistema stereo *in forma standard*. Entrambe le operazioni appena delineate (rimozione della distorsione delle lenti e trasformazione del sistema stereo in forma standard) possono essere descritti mediante delle funzioni o, come nel caso esaminato in questa tesi, mediante delle tabelle. Per maggiori dettagli su come ottenere tali tabelle si rimanda a [1].

La distorsione delle lenti può essere modellata mediante due distinte funzioni:

- ***Tangential Distortion***
  Ovvero l'effetto provocato del disallineamento delle lenti.

- ***Radial Distortion***
  Ovvero l'effetto della non idealità della forma delle lenti.

In alcune applicazioni di computer vision spesso si predilige l'uso di lenti di tipo *wide-angle* o *fisheye*. Per questa tipologia di lenti l'effetto maggiormente visibile è l'effetto *radiale*, che si manifesta curvando le linee che nella scena appaiono dritte, come si può notare dalla figura 1.1. L'entità della distorsione risulta tanto più accentuata quanto più si ci allontana dal centro dell'immagine.

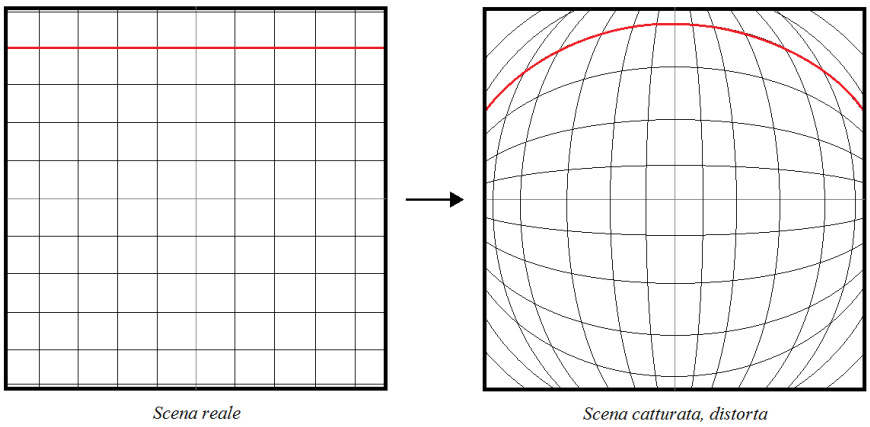


Figura 1.1 fenomeno della distorsione radiale

1.2 Correzione delle distorsioni

Per correggere le distorsioni si può dunque pensare di risalire alla formula matematica che le caratterizza, ovvero la relazione che lega i punti dell'immagine distorta a quelli della non distorta, e dunque di calcolare la nuova coordinata del punto, a run-time, tramite *forward mapping*. Tuttavia questa operazione è difficilmente realizzabile in hardware. Come alternativa possiamo anche pensare di eseguire un *inverse mapping*, ovvero a partire dal punto dell'immagine non distorta, tramite formula inversa, ricavare le coordinate del punto nell'immagine distorta corrispondente.

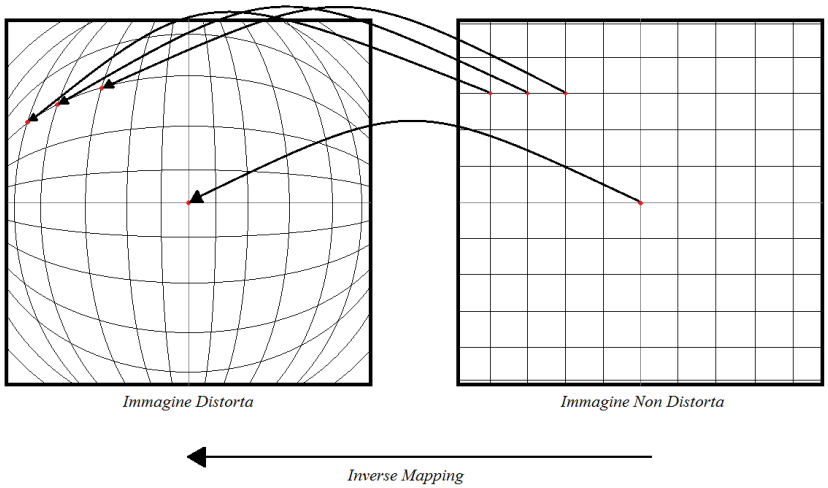


Figura 1.2 Inverse mapping

Questo però non risolve di per sé il problema del costo dell'operazione, ma ci permette di avvicinarci alla soluzione. Possiamo esprimere le distorsioni in termini di *displacement* ovvero lo scostamento del punto dell'immagine non distorta nell'immagine distorta, ed in particolare sulle rispettive coordinate abbiamo:

- *displacement in x*, indicato con *dx*, che indica lo scostamento che devo compiere in ascissa a partire dall'ascissa del pixel dell'immagine non distorta per ottenere l'ascissa del pixel cercato nell'immagine distorta.
- *displacement in y*, indicato con *dy*, che indica lo scostamento che devo compiere in ordinata a partire dall'ordinata del pixel dell'immagine non distorta per ottenere l'ordinata del pixel cercato nell'immagine distorta.

L'idea quindi per realizzare la correzione di queste distorsioni consiste nel precalcolare e salvare i *displacement* sotto forma di tabelle, in modo da limitare la complessità dei calcoli, dovuti all'utilizzo di formule complesse, al minimo indispensabile a tempo di esecuzione. Il metodo migliore per ottenere questi scostamenti, e dunque le tabelle, è effettuare una calibrazione della telecamera, e nel caso di un sistema stereo, di entrambe.

Capitolo 2

Calibrazione

2.1 Calibrazione di una telecamera

Per poter ottenere le matrici dei displacement dunque, bisogna effettuare la calibrazione della telecamera, la quale, come descritto in [2], ci restituirà i parametri intrinseci di distorsione ed estrinseci, nel caso stereo, della telecamera, nonché le matrici dei displacement. Calibrare una telecamera significa dunque determinare i parametri che ci consentono di ottenere un *mapping complessivo* in forma matriciale. Tale procedura si basa sulla individuazione tra punti della scena 3D noti e la loro proiezione 2D. Per ottenere queste corrispondenze si è soliti usare nelle immagini oggetti di forma nota, quali *chessboard*, fig. 2.1, o altri oggetti contenenti *pattern* noti, in modo da rendere il riconoscimento un'operazione semplice. I passi per effettuare una calibrazione sono dunque:

- Acquisizione di una serie di immagini del pattern scelto
- Individuazione di punti particolari della scena facilmente individuabili nell'immagine

- Determinazione parametri telecamera a partire dalla corrispondenza fra punti dell'immagine e punti della scena reale.

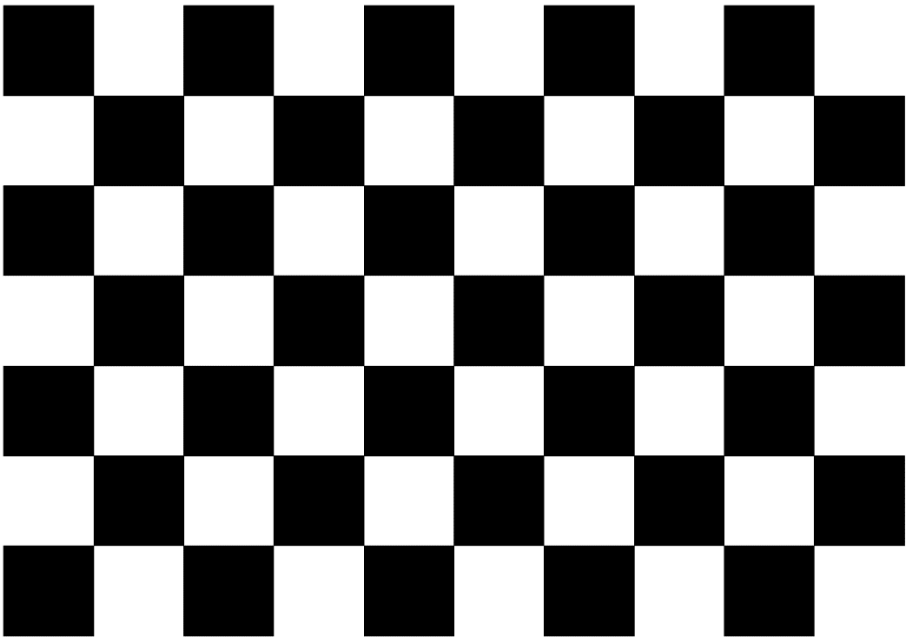


Figura 2.1 Chessboard pattern

2.2 Calibrazione tramite librerie OpenCv

Per effettuare la calibrazione sono state utilizzate le librerie OpenCv, potenti librerie open-source per la computer vision, ed in particolare, tra tutti gli strumenti messi a disposizione, il codice per la calibrazione

messo a disposizione in [3], opportunamente modificato per ottenere il comportamento desiderato. Le funzioni utilizzate nel codice sono [2]:

cv::findChessboardCorners()

Funzione che, data un'immagine monocromatica contenente il pattern scelto, è in grado di individuare e restituire i corners individuati.

cv::drawChessboardCorners()

Funzione che permette di disegnare dei cerchi di vari colori nell'immagine, in corrispondenza dei corners individuati dalla funzione precedente. Ci permette dunque di sapere visivamente se tutti i corners del pattern sono stati individuati.

cv::calibrateCamera()

Funzione che utilizzando i dati appena raccolti su ogni immagine, ci restituisce i parametri intrinseci, che rappresentano la distorsione, ed estrinseci della telecamera, che rappresentano l'orientazione della telecamera nello spazio rispetto al pattern.

cv::fisheye::calibrate()

Nel caso in cui si utilizzassero lenti con elevato grado di distorsione, quali *fisheye* o *wide angle* come nel nostro caso, per ottenere risultati soddisfacenti vi è la necessità di utilizzare questa funzione al posto della *calibrateCamera(),* in quanto per quel tipo di lenti si utilizza un altro modello matematico per la distorsione [2].

A questo punto abbiamo tutti gli elementi per ottenere le mappe dei displacement, che, anche questa volta, otteniamo in modo diverso a seconda del fatto che si usino lenti molto distorte o meno.

cv:: getOptimalNewCameraMatrix()

Tramite questa funzione è possibile decidere, variando il parametro α, se scalare la matrice fondamentale in modo da ottenere un'immagine con più informazione possibile (α = 1), ovvero in cui vengono mantenuti tutti i pixel che si riescono a mantenere, a patto però di inserire delle aree nere, ottenendo un'immagine dai bordi deformi, oppure di ottenere un'immagine ottimale (α = 0), ovvero che elimina qualche pixel utile per eliminare le zone nere e ottenere un immagine rettangolare. Infine si possono ottenere configurazioni intermedie variando α tra 0.0 e 1.0.

cv::initUndistortRectifyMap()

Infine, grazie a questa funzione si ottengono le matrici dei displacement map_x ed map_y.

cv::fisheye::estimateNewCameraMatrixForUndistortRectify()

Equivalente della *getOptimalNewCameraMatrix()* per lenti *fisheye* o *wide angle*.

cv::fisheye::initUndistortRectifyMap()

Equivalente della *initUndistortRectifyMap()* per lenti *fisheye* o *wide angle*.

Capitolo 3

Sistema di telecamere stereo

3.1 Calibrazione di un sistema stereo

La calibrazione di un sistema di telecamere stereo, a differenza di un sistema composto da una singola telecamera, richiede delle funzioni simili alle precedenti, ma non le stesse. Per comprenderne il perché bisogna pensare che, per gli scopi della computer vision, è necessario avere le due telecamere perfettamente allineate, nella cosiddetta *standard form*, dato che, per poter ricostruire la scena in 3D, si vuole che i punti della scena reale si trovino sulla stessa *scanline* nelle due immagini. Mantenere quest'allineamento nella realtà non è cosa semplice/possibile, e quindi, tramite alcune funzioni, è possibile correggere anche questo aspetto, in modo da ottenere alla fine delle immagini corrette, come se fossero state catturate da due telecamere ben allineate. Tramite le librerie OpenCv, ed in particolare il codice messo a disposizione in [4], opportunamente modificato per utilizzare lenti *wide angle* e *fisheye*, è possibile calibrare un sistema stereo. Le funzioni utilizzate nel procedimento sono:

cv::findChessboardCorners() e ***cv::drawChessboardCorners()***

Utilizzate nello stesso identico modo come nel caso di una sola telecamera.

cv::stereoCalibrate()

Funzione che utilizzando i dati appena raccolti su ogni coppia di immagini, ci restituisce i parametri intrinseci, che rappresentano la distorsione, ed estrinseci delle telecamere, che rappresentano l'orientazione relativa delle telecamere nello spazio rispetto a un istema di riferimento opportuno (tipicamente la posizione del pattern durante la prima acquisizione del pattern).

cv::fisheye::stereoCalibrate()

Nel caso in cui si utilizzassero lenti ad elevato grado di distorsione, quali *fisheye* o *wide angle*, come nel nostro caso.

A questo punto per ottenere le mappe dei displacement, tramite la funzione *initUndistortRectifyMap()*, in versione normale o fisheye, c'è bisogno di utilizzare un'altra funzione, in grado di rettificare le due immagini, e cioè di compensare la non idealità del parallelismo tra le due telecamere. In letteratura per fare ciò si usano principalmente due metodi: *Hartley's* [5] e *Bouguet's* [5]. A seconda dell'algoritmo scelto si utilizza una funzione piuttosto che un'altra. Come riportano molti autori, e dati alla mano, il metodo migliore risulta essere quello di *Bouguet* che garantisce un errore minore.

cv:: stereoRectifyUncalibrated()

Funzione utilizzata nel metodo *Hartley*.

cv::stereoRectify()

Utilizzata nel metodo Bouguet. Prepara il tutto per poter ottenere le mappe dei displacement tramite la funzione *initUndistortRectifyMap()*.

cv::fisheye::stereoRectify()

Equivalente della s*tereoRectify()* per lenti *fisheye* o *wide angle*.

Una volta ottenute dunque le quattro mappe dei displacement cercate per il nostro sistema stereo, e cioè left_x, left_y, right_x, right_y, possiamo ritornare sul concetto di *inverse mapping* e concentrarci su questo metodo per effettuare la rettificazione delle singole immagini, intesa come rimozione delle distorsioni e correzione del parallelismo tra le due telecamere.

Capitolo 4

Il processo di rettificazione

4.1 Rettificazione tramite *inverse mapping*

Come già anticipato alla fine del capito precedente, una volta ottenute le quattro mappe, o tabelle, dei displacement, siamo in grado di rettificare le due immagini distorte originali e ottenere quindi delle immagini prive di distorsioni e che abbiamo gli stessi punti della scena reale sulle stesse *scanline*. È bene notare che, in teoria, bisognerebbe dapprima rimuovere le distorsioni tramite un primo mapping, e poi in un secondo momento correggere le prospettive. Nella pratica però si preferisce avere un unico mapping in grado di assolvere a tutti e due i compiti, in modo da poter essere implementato in dispositivi embedded, i quali dovrebbero catturare a 30 e più fps, e quindi per non introdurre ritardi si adotta questo procedimento. Per di più il processo di calibrazione, ovvero l'ottenimento delle matrici, è pensato per essere eseguito *offline* ed è quindi necessario utilizzarlo solamente nel caso in cui ci sia una modifica della geometria del sistema o delle lenti in uso. In fig. 4.1 possiamo vedere il processo di calibrazione, quindi il recupero delle mappe dei displacement, la correzione delle distorsioni

e la rettificazione tramite *pixel mapping* ovvero *inverse mapping* nel nostro caso. Ricordiamo che le due operazioni anche se distinte saranno effettuate insieme tramite un unico mapping.

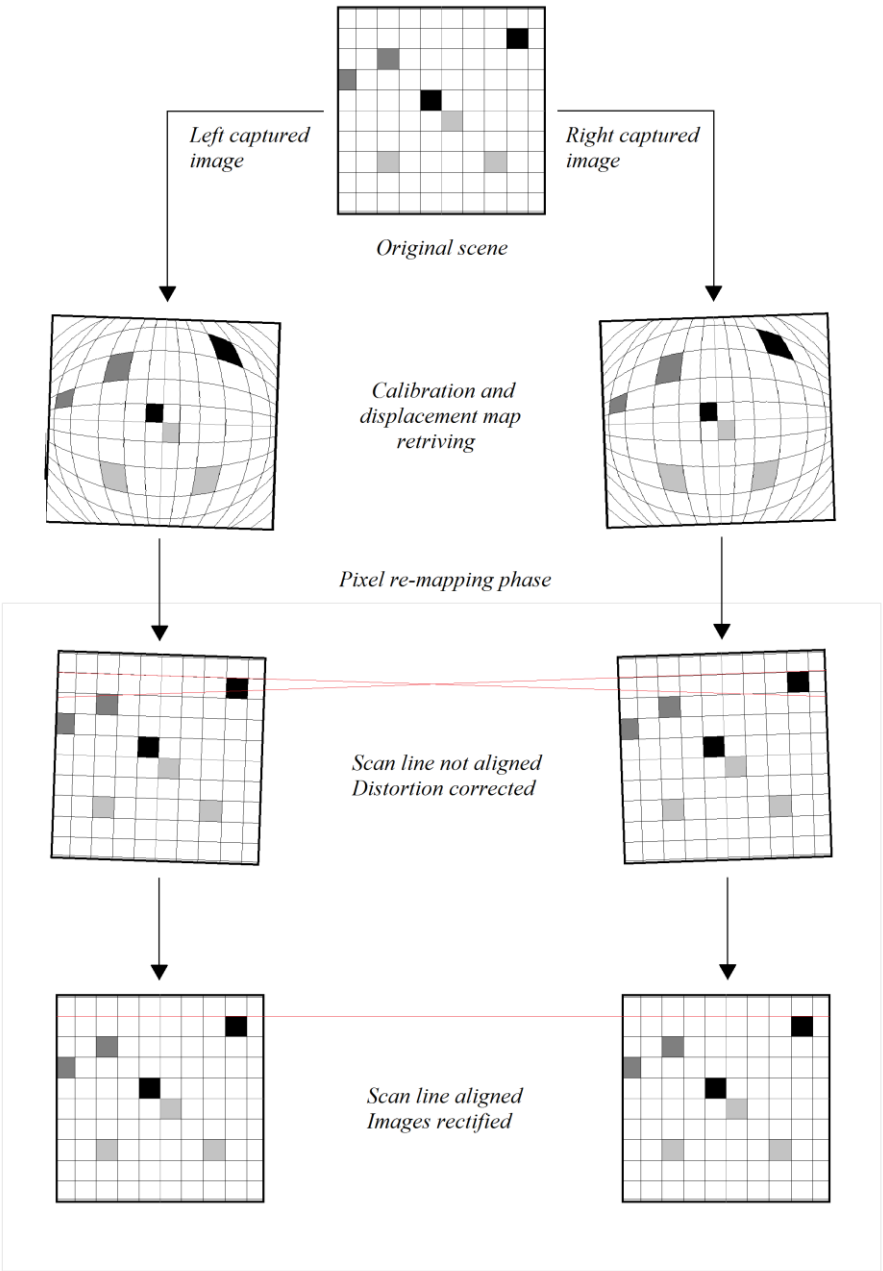


Figura 4.1.1 Processo di calibrazione e rettificazione

Analizziamo ora più da vicino l'inverse mapping, per scoprire un problema secondario:

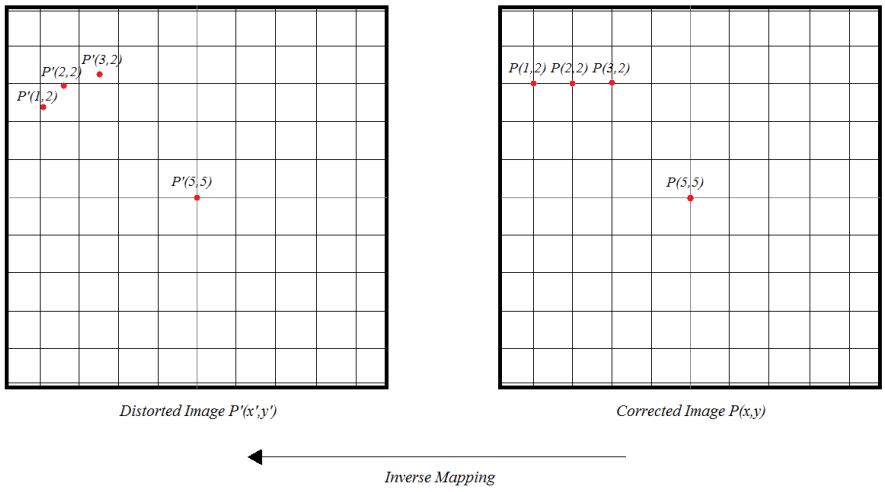


Figura 4.1.2 Problemi nel recupero pixel tramite mapping, in ogni angolo dei quadrati abbiamo un pixel reale.

Ogni punto della *Corrected Image* ha un corrispondente nella *Distorted Image*. Questa informazione è memorizzata in forma assoluta nelle due matrici dei displacement, ovvero, per una determinata coordinata (x,y) della *Corrected* image, si avrà che alla coordinata (x,y) della tabella delle x, avrò la coordinata assoluta x', e della tabella delle y, avrò la coordinata assoluta y' del pixel della *Distorted Image* da prendere e inserire alla coordinata (x,y) della *Corrected Image*. I valori dunque memorizzati nelle tabelle sono assoluti e di tipo float, ma volendo, per risparmiare memoria in termini di bit, possono essere trasformati in relativi sottraendo al valore memorizzato l'ascissa o l'ordinata, a seconda del fatto che sia una tabella x oppure y, della coordinata a cui si trova tale valore. Questo però fa sì che si debba fare un'operazione in più - una somma – affinché si possa recuperare correttamente l'ascissa e l'ordinata da cui ricavare il pixel nella *Distorted Image*. Ci

accorgiamo a questo punto però che, a causa della natura continua della distorsione [2], non tutti i pixel della *Corrected Image* puntano a un preciso pixel della *Distorted Image.* Dobbiamo dunque determinare l'intensità, ovvero il valore, del pixel da associare in modo *approssimato*.

4.2 Interpolazione dell'intensità del pixel

Uno dei metodi maggiormente utilizzati in letteratura [5] per interpolare l'intensità del pixel desiderato, per via della sua semplicità e il basso costo computazionale, è l'interpolazione bilineare. Questa tecnica consiste nell'utilizzare i quattro pixel adiacenti al punto di coordinate float che cerchiamo in modo pesato, per calcolare un valore approssimativo dell'intensità del pixel nel punto cercato. L'interpolazione bilineare si definisce in genere come:

$$I = \frac{aA + bB + cC + dD}{a + b + c + d}$$

Dove i pesi sono calcolati come:

$$a = (1 - x)(1 - y)$$
$$b = x(1 - y)$$
$$c = (1 - x)y$$
$$d = xy$$

Nel nostro caso, con riferimento alla seguente figura, invece

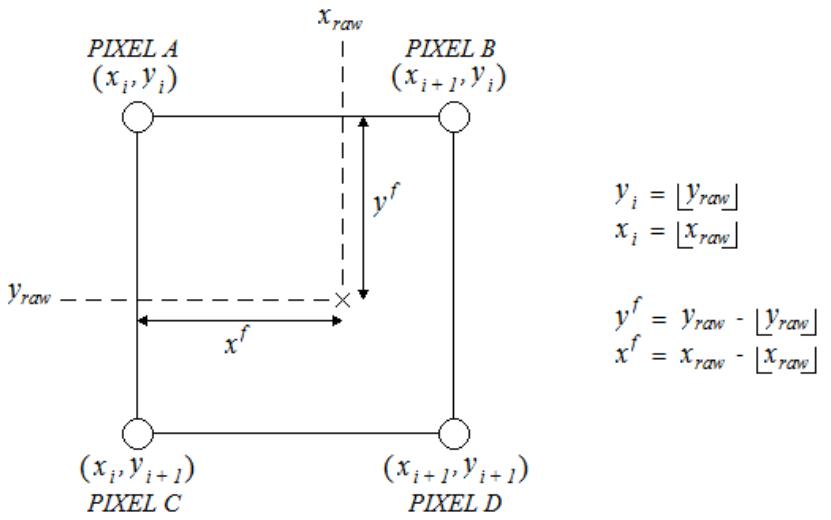


Figura 4.2.1 Interpolazione bilineare dell'intensità nella Distorted Image

Indichiamo con x_{raw} e y_{raw} le coordinate del punto cercato nella *Distorted Image*, con x_i e y_i le coordinate del pixel in alto a sinistra ottenute dalla parte intera rispettivamente di x_{raw} e y_{raw}, con x^f e y^f la distanza in x e in y di x_{raw} e y_{raw} dal punto di coordinate x_i e y_i ed infine con x_u e y_u le coordinate del punto della *Corrected Image* di cui si vuole ottenere l'intensità del punto di coordinate x_{raw} e y_{raw} della *Distorted Image*, si ha che, a meno della normalizzazione per la somma dei pesi, l'intensità è calcolata come:

$$
\begin{aligned}
I^{rect}_{x_u,y_u} = \; & I^{raw}_{x_i,y_i}(1-x^f)(1-y^f) \\
& + I^{raw}_{x_{i+1},y_i}\, x^f(1-y^f) \\
& + I^{raw}_{x_i,y_{i+1}}\,(1-x^f)y^f \\
& + I^{raw}_{x_{i+1},y_{i+1}}\, x^f\, y^f
\end{aligned}
$$

Scegliere di utilizzare solo i quattro pixel adiacenti, pesati, per interpolare il valore dell'intensità, risulta essere un buon compromesso tra accuratezza e semplicità dell'operazione, ma volendo è possibile utilizzare un numero di pixel sempre maggiore, realizzando un'interpolazione sempre più accurata e *smooth*, ma sempre più costosa, chiaramente. In alternativa è invece possibile scegliere di non interpolare affatto l'intensità, ma semplicemente di scegliere il pixel reale più vicino alle coordinate del punto (x_{raw}, y_{raw}) tramite *round*. Questo comporta però un errore elevato, constatabile dal fatto che i contorni degli oggetti nell'immagine, prima continui, diventano *seghettati*. Mostriamo ora i risultati ottenibili in entrambi i casi.

4.3 Rettificazione con e senza interpolazione

Per semplicità mostreremo solo i risultati ottenuti rettificando tramite prototipo l'immagine Left, con e senza interpolazione, poiché si ottengono risultati analoghi usando l'immagine Right. Il *dataset* utilizzato nell'esempio in questione, utilizza lenti che introducono distorsione, ma non di tipo *wide angle*, e verrà poi sostituito da un *dataset* che utilizza lenti *wide angle*, per mostrare anche l'aumento di distorsione che si ha con questo tipo di lenti.

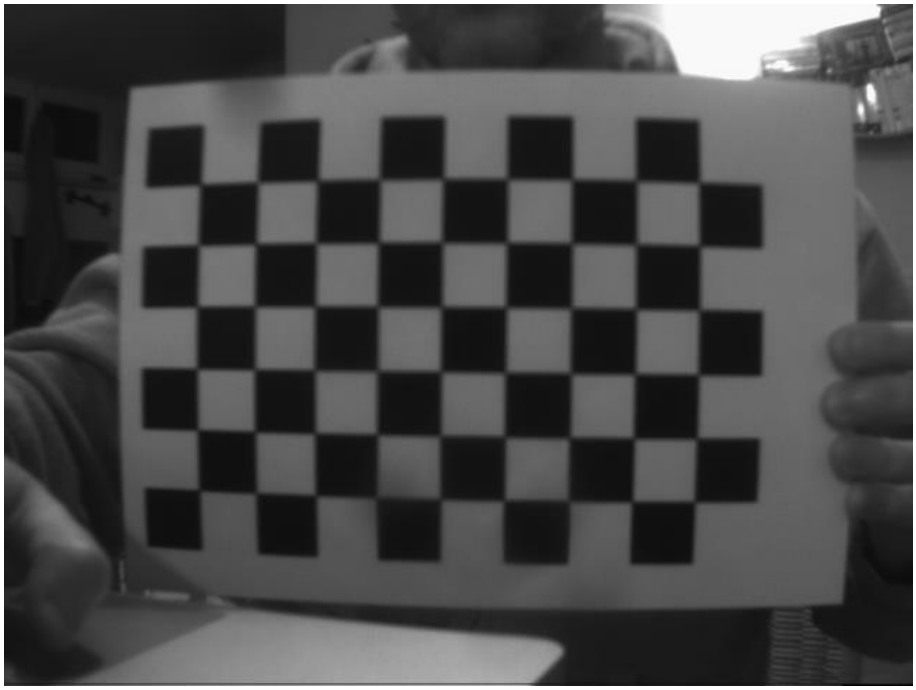


Figura 4.3 Immagine Left di riferimento, Distorta, da rettificare

4.3.1 Rettificazione *senza* interpolazione

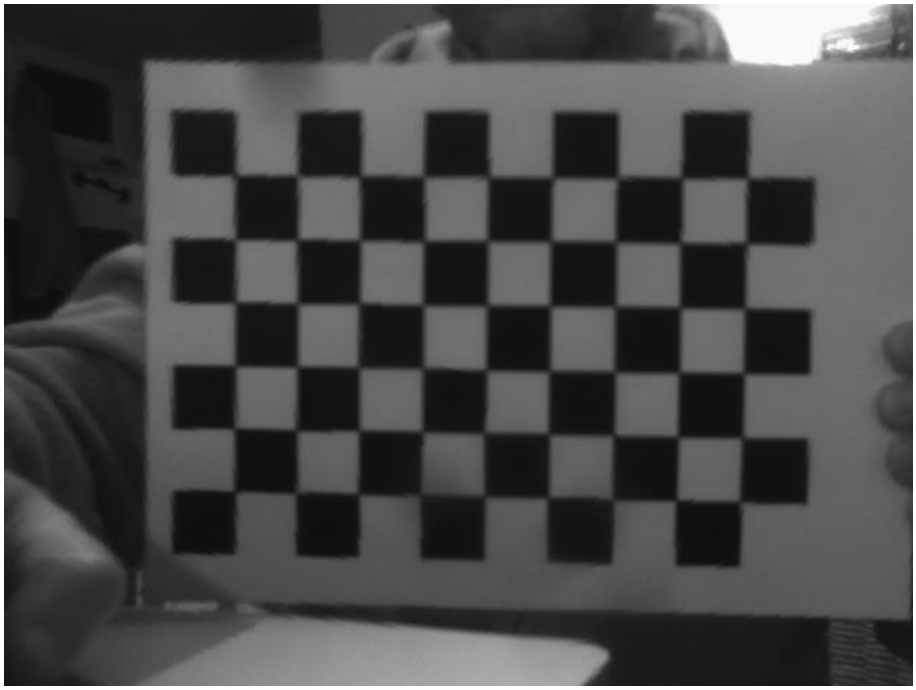


Figura 4.3.1 Immagine Left rettificata senza interpolazione

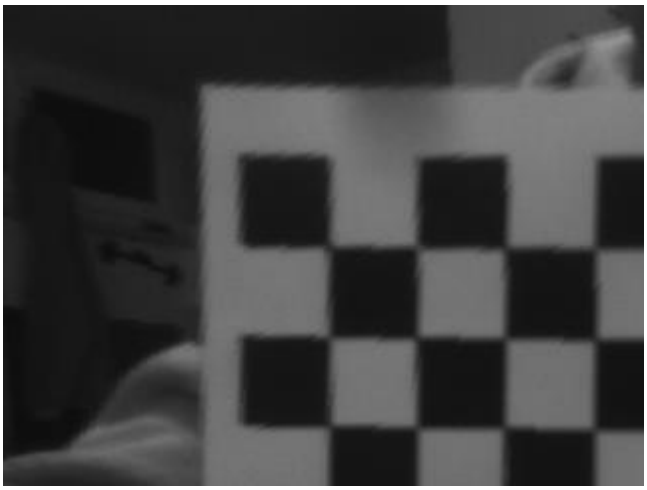


Figura 4.3.1.1 Dettagli Immagine Left rettificata senza interpolazione

4.3.2 Rettificazione *con* interpolazione bilineare float

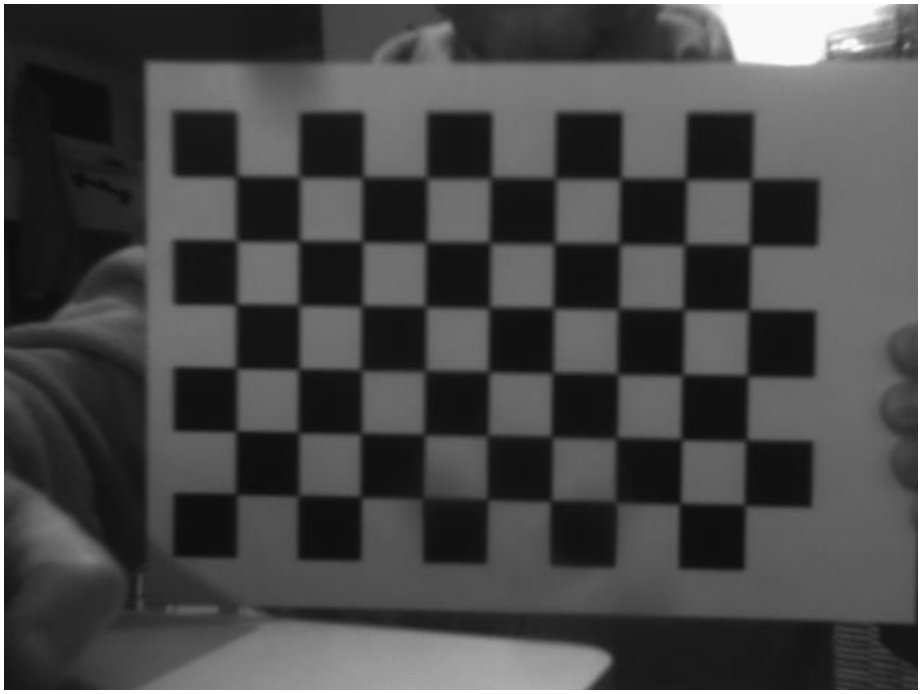


Figura 4.3.2 Immagine Left rettificata con l'interpolazione bilineare e l'uso dei float

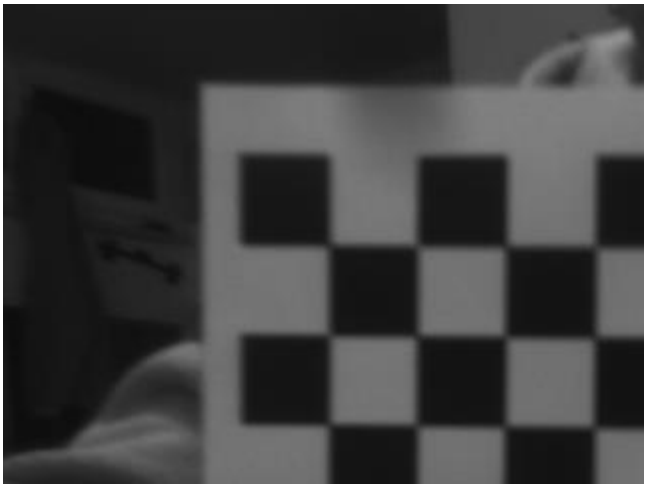


Figura 4.3.2.1 Dettagli Immagine Left rettificata con l'interpolazione bilineare e l'uso dei float

Come si evince dai risultati, la rettificazione senza interpolazione introduce nell'immagine artefatti molto evidenti, che rendono inusabile tale soluzione. I risultati della rettificazione float con interpolazione bilineare, invece, rappresentano un ottimo compromesso tra prestazioni e qualità dell'immagine ottenuta, e, come vedremo in seguito, rappresenta lo standard da raggiungere una volta applicate le semplificazioni necessarie per realizzare la rettificazione nei dispositivi embedded, come ad esempio le FPGA.

Capitolo 5

Rettificazione fixed-point

5.1 Dal floating-point al fixed-point

Come anticipato in precedenza, per far sì che la rettificazione possa essere portata su dispositivi embedded, vi è la necessita di introdurre dei cambiamenti e delle semplificazioni al metodo originale. La prima semplificazione che si deve adottare, quasi necessariamente, è il passaggio dal tipo di dato *floating-point*, disponibile solo in alcuni dispositivi, ad un tipo certamente disponibile anche nei dispositivi di più bassa caratura, e cioè il *fixed-point*. Ciò è sicuramente vero in quanto di fatto il *fixed-point* si basa sugli interi, disponibili praticamente in tutti i dispositivi. Ma andiamo con ordine, quello che vogliamo fare è dare una rappresentazione ai numeri *con la virgola* usando i numeri *interi*. In che modo possiamo fare ciò? Nella versione implementa su PC, semplicemente utilizzando una certa quantità di bit di un numero *integer* per rappresentare la parte *decimale* del numero float che vogliamo rappresentare, e i rimanenti bit per rappresentare la parte intera del medesimo numero float. Stiamo dunque passando da un dato di tipo *floating-point*, ovvero a virgola mobile, ad un tipo di dato *fixed-*

point, ovvero a virgola fissa. Quindi, a differenza dei *floating-point*, il *fixed-point* è *una coperta corta*: tanto più si aumenta la precisione della parte decimale, e dunque il numero di bit che si riservano a quest'ultima, tanto più aumenta la probabilità di andare in *overflow*, dato che il numero di bit totali è fisso e dunque la differenza tra i bit totali dedicati ad un intero, e i bit che riserviamo per la parte decimale determina il numero massimo rappresentabile dalla parte intera prima di andare in *overflow*. Nel nostro caso però, il compromesso non è un problema, in quanto andremo ad utilizzare il *fixed-point* per rappresentare dei displacement, che anche nel caso fossero memorizzati per intero, non supererebbero la dimensione della *width* o della *height* dell'immagine da rettificare. Verrebbe utilizzato poi per l'interpolazione bilineare, ma anche qui, dalla formula, il valore massimo teorico sarebbe 255 + 255 + 255 + 255 = 1020, ma per come è strutturata la formula, il numeratore non dovrebbe mai superare 255, ed il denominatore 1, il che ci lascia un ampio margine per la parte decimale. Vediamo dunque nel dettaglio come procedere.

5.2 Conversione floating-point - fixed-point

Per convertire un numero da floating-point a fixed-point dobbiamo innanzitutto decidere la quantità di bit da riservare per la parte decimale, che indicheremo con *nbit*. Una volta scelta questa quantità ci basterà *moltiplicare* il numero float per 2^{nbit}, ovvero, più semplicemente, effettuare *nbit* shift a sinistra della quantità float, ed infine *castare* ad intero – con o senza *round* – il valore ottenuto. Mentre per effettuare il contrario, ci basterà *dividere* il numero fixed-point per 2^{nbit}, ovvero effettuare *nbit* shift a destra della quantità fixed-point, opportunamente assegnata ad un float per non perdere i decimali. Ovviamente non vi è

alcuna necessità di utilizzare le potenze di due per convertire i numeri, dato che possiamo utilizzare un qualunque numero, ed otterremmo l'effetto desiderato. *Conviene* però utilizzare le potenze di due, in quanto in hardware, ma spesso anche in software, la divisione e la moltiplicazione tra interi, di numeri in cui il moltiplicatore o il divisore è una potenza di due, viene effettuata da un veloce shift, al posto della lenta divisione o moltiplicazione. Andiamo ora ad analizzare cosa implica però il passaggio da una forma all'altra.

5.3 Perdita d'informazione nella conversione

Come suggerito dal titolo, il problema principale nell'utilizzare il fixed-point è la perdita d'informazione rispetto ai float. Se consideriamo l'implementazione dei float però, scopriamo che ovviamente anch'essi hanno problemi nel rappresentare i numeri reali, dovuti all'utilizzo del codice binario per rappresentarli. Ad esempio sappiamo che 0.1 in float non è lo stesso numero reale che conosciamo noi ma qualcosa di molto vicino ad esempio 0.100000001. Tutto normale, dato che ad esempio nel nostro sistema a base dieci abbiamo difficoltà a rappresentare 1/3. Questo significa, infatti, che per ogni base esistono dei numeri che non sono rappresentabili con un numero finito di cifre, e quindi, sia i float, che i fixed-point, non li rappresenteranno mai alla perfezione. Senza approfondire troppo ricordiamo che i float sono rappresentati da un numero, con segno, e da un altro numero con segno che rappresenta l'esponente della potenza di dieci da moltiplicare al numero, una volta sommatovi 1, per ottenerlo nella forma abituale, praticamente dove mettere la virgola, e che l'errore dipende dalla magnitudo, ovvero i numeri troppo grandi e i numeri troppo piccoli non sono rappresentabili. A questo punto sappiamo che i float, ad una certa cifra, approssimano i

numeri reali, e che i fixed-point approssimano i numeri float – e dunque i reali – ad un'altra cifra. Vediamo meglio dall'immagine seguente cosa succede.

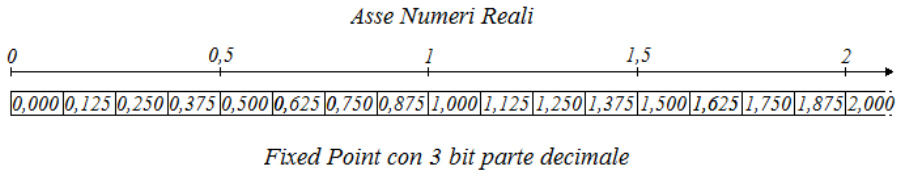


Figura 5.3 Relazione tra rappresentazione reali e fixed-point

Un numero float a 32 bit può rappresentare numeri che vanno da 3.4×10^{38} a -3.4×10^{38} ed il numero più piccolo che può rappresentare è $1.175494351 \times 10^{-38}$, cioè divide i numeri reali a blocchi di grandezza $1.175494351 \times 10^{-38}$, e questo significa che le cifre o i numeri più piccoli di quello minimo vengono approssimati al numero multiplo di quello minimo più vicino, in base alla presenza di round o meno. Con i fixed-point accade la stessa cosa, e quindi usando Int a 32 bit, e riservando 3 bit per la parte decimale, dalla fig. 5.3 si ha che i numeri decimali vengono divisi in 8 blocchi da 0.125 e quindi in base al round o meno un numero viene approssimato al multiplo di 0.125 più vicino. Convertiamo per esempio 23.719381 da float a fixed-point con 3 bit, otteniamo $23.719381 \times 8 = 189.755048$, da portare ad intero, dunque nel caso in cui utilizzo le normali regole di *round* diventa 190. Se ora lo riconvertiamo in float abbiamo 190 / 8 = 23.750 che differisce dal numero originale di 0.030619. La soluzione per ridurre la differenza tra rappresentazione float e fixed-point ovviamente consiste nell'aumentare il numero di bit da usare per la parte decimale, diminuendo così sempre di più la dimensione degli intervalli che rappresentano i numeri reali, tenendo bene a mente però che non vi

potrà mai essere una precisione assoluta, o comunque migliore dei float a parità di bit.

5.4 Operazioni tra fixed-point

Le operazioni tra fixed-point con lo stesso numero di bit per la parte decimale sono le stesse degli interi quindi gli unici pericoli sono l'*overflow* dovuta a somme, sottrazioni e moltiplicazioni, e la possibile perdita di decimali dovuta alla divisione tra interi. La somma e la sottrazione non hanno altri problemi, escludendo la somma e la sottrazione tra un fixed-point e un intero o un float, che richiede prima che l'intero o il float sia trasformato in fixed-point. Mentre la moltiplicazione porta con sé un altro problema: un fixed-point è fondamentalmente un numero intero moltiplicato un altro numero che rappresenta la discretizzazione dell'asse reale: possiamo dunque pensarlo come $fixed = int \times discr\_factor$. Se moltiplichiamo due fixed-point otteniamo quindi che $multiplied\_fixed = int1 \times int2 \times discr\_factor \times discr\_factor$, con un rischio altissimo di *overflow* e che non è più nella stessa base di prima. Dobbiamo quindi effettuare una divisione per riportare nella vecchia base. Il problema invece si risolve da sé moltiplicando per un int o un float. Nella divisione invece abbiamo il problema opposto, ovvero il *discr_factor* scompare dividendo due fixed-point, con l'evidente risultato di aver perso i decimali. La soluzione è quindi di effettuare una moltiplicazione del numeratore per il *discr_factor* prima di effettuare la divisione, che mantiene certamente i decimali, ma potrebbe causare anche un overflow. Come prima la divisione per un float o un int non da problemi. C'è anche da considerare un'altra cosa: il segno del numero fixed-point. In software non vi è alcun problema, in quanto i compilatori

moderni, infatti, mantengo il segno mentre effettuano uno shift, mentre in hardware non è detto che ci sia questa *feauture* e quindi va considerata. Infine nel caso di fixed-point con basi diverse vi è la necessità di convertire uno dei due nella base dell'altro, operazione facilmente effettuabile moltiplicando per la base dell'altro e poi dividendo per la propria base il numero da convertire.

Una volta chiariti questi aspetti siamo in grado di implementare la rettificazione con l'utilizzo dei fixed-point.

5.5 Rettificazione con fixed-point

Per implementare la rettificazione in fixed-point vi è bisogno, in ultima analisi, di modificare la formula per l'interpolazione bilineare in modo da poter essere utilizzata con quest'ultimi. In particolare: x_i e y_i non saranno più ottenute dal cast a int di x_{raw} e y_{raw}, che ora sono dei fixed-point, ma dalla divisione, intesa sempre come shift, per 2^{nbit}. x^f ed y^f ora sono ottenute facendo l'& con una maschera che prenderà solo i bit dedicati alla parte decimale. Infine, il numero "1" che compare nella formula viene dunque portato in fixed-point e quindi il nuovo "1" sarà $2^{nbit} \times 1$. Una costante, che qui chiameremo 1_*fixed*. Per la natura della formula non vi è nemmeno il bisogno di correggere la moltiplicazione tra l'intensità, intera, e i due fixed-point, in quanto dopo si dividerà per la somma dei pesi, che è un prodotto di fixed-point sistemando il tutto. L'ultima accortezza si ha nel caso dei displacement relativi, in quanto si dovrà convertire la coordinata x o y in fixed-point, prima di sommarla al displacement per ottenere la coordinata del pixel nell'immagine distorta.

Precisato ciò la formula diventa:

$$
\begin{aligned}
I^{rect}_{x_u,y_u} = \; & I^{raw}_{x_i,y_i}(1\_fixed - x^f)(1\_fixed - y^f) \\
& + I^{raw}_{x_{i+1},y_i}\, x^f (1\_fixed - y^f) \\
& + I^{raw}_{x_i,y_{i+1}}\, (1\_fixed - x^f) y^f \\
& + I^{raw}_{x_{i+1},y_{i+1}}\, x^f\, y^f
\end{aligned}
$$

Mostriamo dunque ora il migliore, e il peggior risultato ottenuto dalla rettificazione con interpolazione bilineare fixed-point del *dataset wide angle*, avendo posto come limite nei test 10 bit per la parte decimale, rimandando il lettore al capitolo sui risultati sperimentali per una disamina più completa.

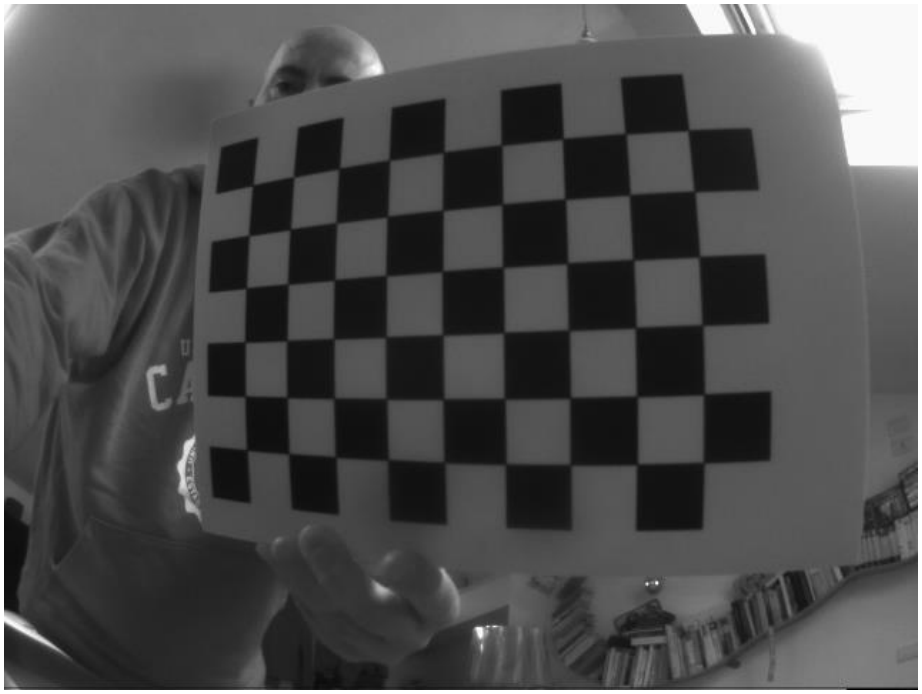


Figura 5.5.1 Immagine Left da rettificare di riferimento

Utilizzando 10 bit si ottengono risultati visivamente equivalenti alla versione float:

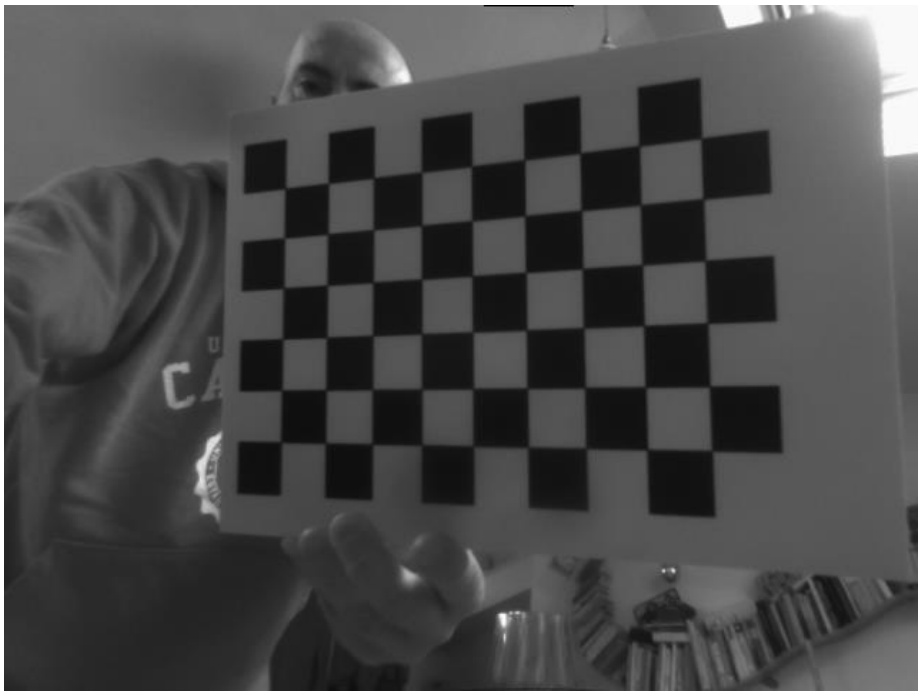


Figura 5.5.2 Immagine Left rettificata con l'interpolazione bilineare e l'uso dei fixed-point con 10 bit per la parte decimale.

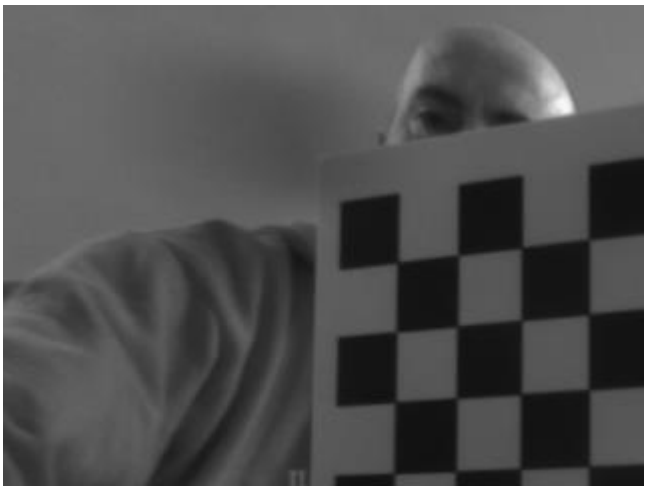


Figura 5.5.3 Dettagli immagine Left rettificata con l'interpolazione bilineare e l'uso dei fixed-point con 10 bit per la parte decimale.

Utilizzando invece 1 bit solo si ha ovviamente il risultato peggiore, simile visivamente alla versione senza interpolazione:

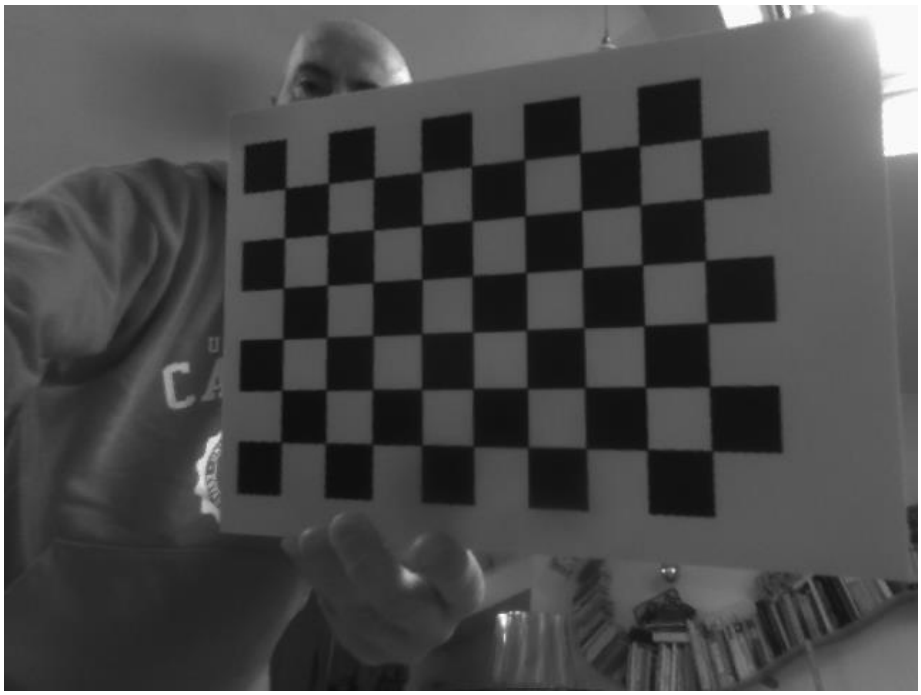


Figura 5.5.4 Immagine Left rettificata con l'interpolazione bilineare e l'uso dei fixed-point con 1 bit solo per la parte decimale.

Figura 5.5.5 Dettagli immagine Left rettificata con l'interpolazione bilineare e l'uso dei fixed-point con 1 bit solo per la parte decimale.

Capitolo 6

Rettificazione con riduzione delle matrici dei displacement

6.1 Dimensioni delle matrici dei displacement

Arrivati a questo punto siamo in grado di rettificare, in hardware, con un procedimento poco costoso in termini di risorse computazionali o hardware dedicato implementabili praticamente in ogni dispositivo embedded esistente. Tuttavia, è facilmente verificabile che memorizzare per intero le quattro matrici dei displacement necessarie per la rettificazione stereo, richiederebbe un quantitativo di memoria non indifferente e proporzionale alla dimensione delle immagini, non sempre disponibile, in particolare nelle FPGA. In quest'ultime infatti la quantità di memoria disponibile all'interno del dispositivo è una delle specifiche che influisce maggiormente sul costo. Essendoci posto il limite di utilizzare dispositivi il più economico possibile, si è valutato, come descritto anche in [5], una metodologia che consenta di ridurre il quantitativo di memoria richiesta. Tuttavia, anche nel caso di matrici relative memorizzare quattro matrici delle dimensioni delle immagini,

quindi 640x480, utilizzando ad esempio un fixed-point a 16 bit, per esempio, con 6 bit per la parte decimale richiederebbe, per tutte e quattro le matrici, 2457600 Byte, e cioè quasi 3 Mbyte.

6.2 Riduzione delle matrici dei displacement

La soluzione proposta è dunque quella di ridurre la dimensione delle matrici dei displacement di un certo fattore di riduzione, detto *reduction_factor*, e poi di ricostruire i valori mancanti dalle tabelle a *run-time* tramite un'interpolazione bilineare. Quello che succede nella pratica dunque è che la matrice dei displacement è *campionata*, a partire da 0, ogni *reduction_factor*, e le dimensioni finali quindi, in *width* e *height*, si riducono del *reduction_factor* scelto. Ad esempio scegliendo le nostre immagini con dimensione 640x480 avremo ovviamente quattro matrici da 640x480, che ridotte di un *reduction_factor* = 16 diventerebbero delle dimensioni di 640/16 = 40, 480/16 = 30 e quindi 40x30, e utilizzando lo stesso esempio di prima del fixed-point a 16 bit otterremmo 9600 Byte, e cioè quasi 10 Kbyte, ovvero quasi 300 volte meno. Questa riduzione però ha un costo, ovvero si introduce un errore nelle coordinate del punto da interpolare, ovvero x_{raw} e y_{raw}, che viene dunque mal posizionato prima di effettuare l'interpolazione dell'intensità. Quest'errore inoltre, sarà tanto maggiore quanto più ci si allontana dal centro ottico dell'immagine, a causa della distorsione di tipo *radiale* che caratterizza l'uso di lenti, soprattutto quelle di tipo *wide angle e fisheye*. Bisogna poi introdurre una nova rete in grado di effettuare questa operazione, aggiungendo complessità al progetto finale. Essa però rappresenta una delle poche alternative realmente disponibili atte a risolvere il problema della mancanza di memoria, e si sceglie quindi di utilizzare tale metodo anche a fronte dell'errore,

cercando però di mantenerlo il più basso possibile. Vediamo meglio l'effetto della riduzione delle matrici con delle figure.

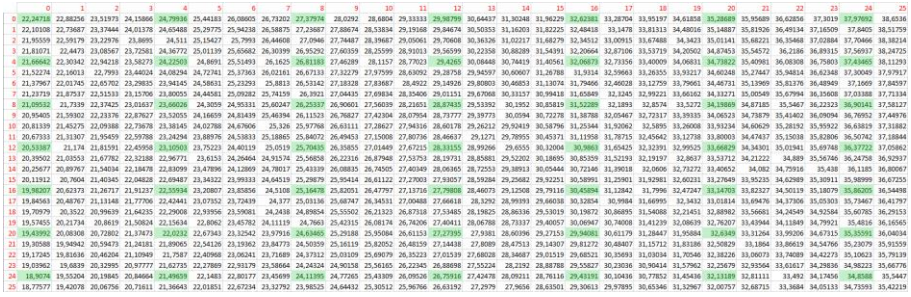


Figura 6.2.1 Valori campionati riducendo una matrice dei displacement con reduction_factor 4

Come anticipato quello che succede è che vengono presi solo determinati valori dalla matrice originale, in verde nella fig. 6.2.1, ottenendo così una matrice ridotta.

	0	1	2	3	4	5	6
0	22,24718	24,79936	27,37974	29,98799	32,62381	35,28689	37,97692
1	21,66642	24,22503	26,81183	29,4265	32,06873	34,73822	37,43465
2	21,09532	23,66026	26,25337	28,87435	31,52289	34,19869	36,90141
3	20,53387	23,10503	25,70435	28,33155	30,9863	33,66829	36,37722
4	19,98207	22,55934	25,16478	27,79808	30,45894	33,14703	35,86205
5	19,43992	22,0232	24,63465	27,27395	29,94081	32,6349	35,35591
6	18,9074	21,49659	24,11395	26,75916	29,43191	32,13189	34,8588

Figura 6.2.2 Valori float campionati e salvati nella matrice ridotta riducendo una matrice dei displacement con reduction_factor 4

Convertiamo i valori in fixed-point con 4 bit per la parte decimale:

	0	1	2	3	4	5	6
0	356	397	438	480	522	565	608
1	347	388	429	471	513	556	599
2	338	379	420	462	504	547	590
3	329	370	411	453	496	539	582
4	320	361	403	445	487	530	574
5	311	352	394	436	479	522	566
6	303	344	386	428	471	514	558

Figura 6.2.3 Valori della matrice ridotta con fixed-point a 4 bit per la parte decimale

Ogni cella rappresenta quindi un'area di 4 displacement e quindi significa che mentre scorriamo l'immagine da rettificare dividiamo le coordinate (x_u, y_u) per il *reduction_factor*, ottenendo quindi, dalla divisione tra interi, le coordinate del displacement corrispondente e quindi dei 3 displacement adiacenti. Effettuiamo, infine, un'interpolazione bilineare per ricavare il displacement mancante tra i quattro a disposizione. Nel caso poi la matrice sia relativa sommiamo al valore ottenuto rispettivamente x_u per ricavare la coordinata x, e y_u per la coordinata y.

6.3 Rettificazione con matrici ridotte ed effetti di bordo

Se effettuiamo quindi le operazioni descritte in precedenza, siamo in grado di effettuare la rettificazione delle immagini, ma ci troviamo ad avere a che fare con un nuovo inconveniente. Le immagini rettificate, infatti, presentano, in prossimità del bordo destro e di quello inferiore, delle aree nere, della dimensione del *reduction_factor* utilizzato, introdotte a causa del fatto che l'ultima riga e l'ultima colonna della matrice ridotta sono di fatto l'ultima riga e colonna che è possibile prendere campionando ogni *reduction_factor*. Facciamo un esempio. Se riduciamo una tabella 640x480 di un *reduction_factor* 32 otteniamo una matrice 20x15. L'elemento numero 20 della riga 1 è l'elemento (19, 0), dato che gli indici vanno da 0 a 19, e da 0 a 14, quindi significa che il valore campionato in questa cella è il valore che è nella posizione 19 × 32 = 608 dunque (608, 0), che è l'ultimo elemento che è possibile prendere dato che l'elemento successivo sarebbe nella posizione (640,0), che non esiste dato che la matrice originale va da 0 a 639, ovvero 640 colonne, e da 0 a 479, ovvero 480 righe. A causa di questo motivo per i pixel dell'immagine che si trovano tra l'ultimo elemento campionabile, nel nostro esempio 608, ed il bordo dell'immagine, non si dispone di tutti e quattro i displacement necessari per effettuare un'interpolazione bilineare, e quindi si marcano tali pixel come neri. L'area utile dell'immagine viene dunque ridotta. Vediamo ora dunque un esempio di rettificazione tramite fixed-point e riduzione delle matrici dei displacement.

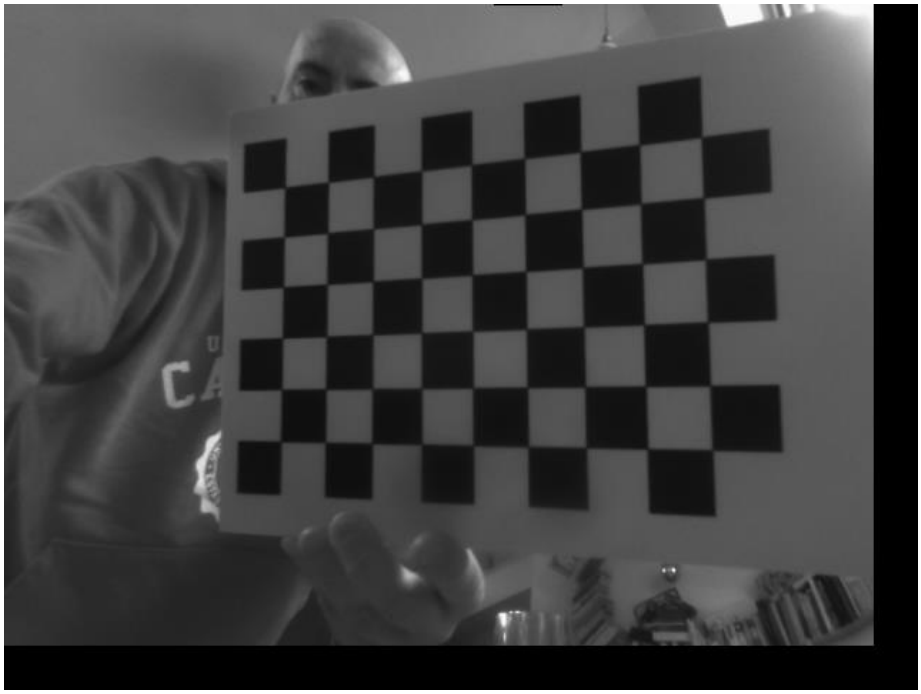


Figura 6.3.1 Effetti di bordo nell'immagine Left rettificata con interpolazione bilineare, 4 bit per i fixed-point e reduction_factor 32

6.4.1 Prima soluzione agli effetti di bordo

Una prima soluzione adottata è stata quella di aggiungere una riga ed una colonna alle tabelle ridotte, e di inserivi, in qualunque caso, l'ultimo elemento corrispondente e disponibile nella matrice originale. Ovvero per l'ultima colonna della matrice ridotta, riprendendo l'esempio di prima 640x480, prenderemo gli elementi alla 639 colonna e per l'ultima riga prenderemo gli elementi alla 479 riga. Questo procedimento, anche se introduce un errore non trascurabile nelle aree precedentemente visualizzate in nero, ci permette di avere di nuovo il 100% dell'area utile nell'immagine. Vediamo la stessa immagine con gli stessi parametri rettificata tramite questa prima soluzione.

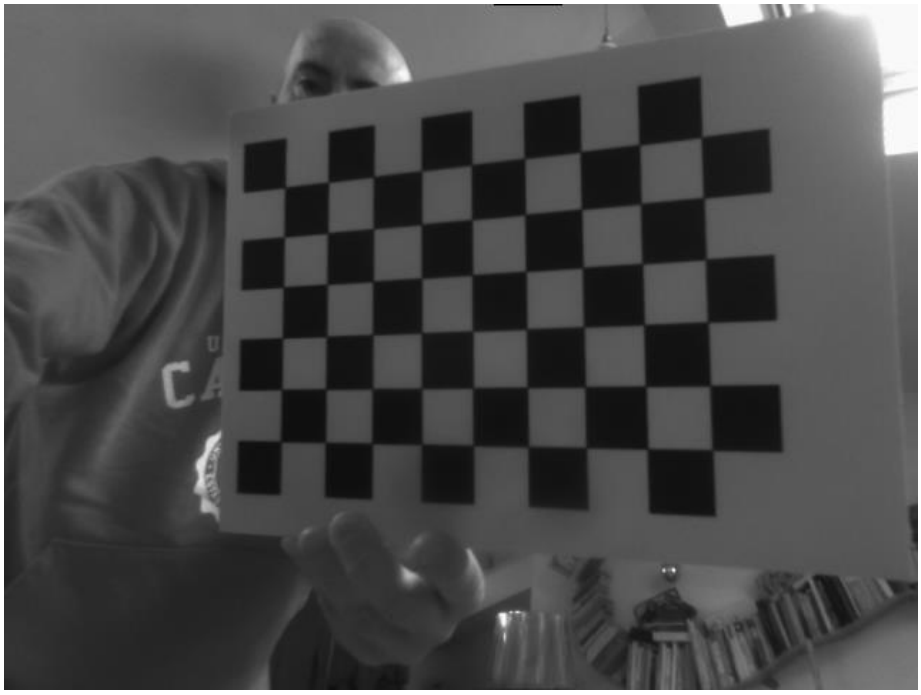


Figura 6.4.1 Effetti di bordo assenti nell'immagine Left rettificata con interpolazione bilineare, 4 bit per i fixed-point e reduction_factor 32 e metodo 1

Questo metodo però, come anticipato prima, introduce un errore elevato nei bordi, e per vederlo e quantificarlo al meglio mostriamo ora *l'RMSE, Root Mean Square Error*, congiunto delle matrici dei displacement *left_x* e *left_y*, riferito alla medesima configurazione dell'immagine precedente.

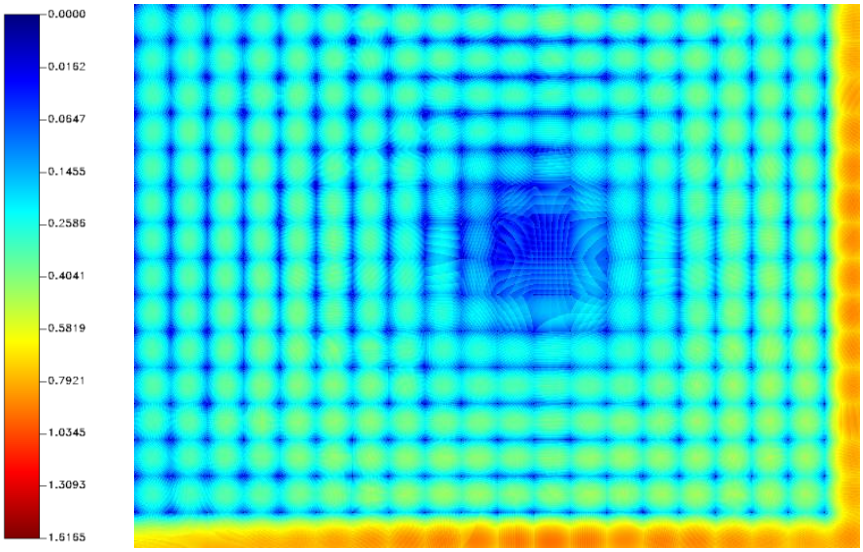


Figura 6.4.1.1 Effetti di bordo nell'immagine Left rettificata non visibili ad occhio nudo, ma evidenti e quantificabili tramite analisi numerica con scala non lineare

Rimandando il lettore al capitolo sui risultati sperimentali per una disamina più completa *dell'RMSE* nelle varie tipologie di rettificazione, intuiamo molto presto che questo metodo non è né il più efficiente né il migliore, dato che ci permetti di ottenere un'immagine completa sì, ma potenzialmente inutilizzabile in tali aree per gli scopi finali del progetto, ovvero il calcolo delle corrispondenze stereo [6].

6.4.2 Seconda soluzione agli effetti di bordo

Come è stato appena evidenziato, la prima soluzione proposta per la gestione degli *effetti di bordo* non è soddisfacente. Scegliendo di inserire come ultima riga e ultima colonna proprio quelle immediatamente precedenti a quelle necessarie, che ovviamente non esiste, perturba significativamente l'interpolazione nelle regioni

estreme in basso e a destra. In effetti, il metodo precedente, non considera un aspetto fondamentale, ovvero che il valore che ci servirebbe è proprio quello successivo a quello disponibile. Questa strategia è però critica, in quanto, in questo modo i campioni delle matrici sono trattati come se fossero indipendenti gli uni dagli altri, anche se questo non è vero. C'è infatti una certa dipendenza statistica tra i campioni delle matrici, dovuta al fatto che a generare questi valori non è il caso, ma una formula ben definita, che, come molte formule, varia nei valori in modo graduale. Studiando questa correlazione siamo arrivati alla conclusione che, per esempio, gli elementi delle tabelle delle x, scorrendo le colonne, aumenta il proprio valore in modo graduale, e generalmente, tende ad incrementare la quantità dell'aumento in modo lento, mantenendo l'aumento *costante* verso il centro ottico dell'immagine e meno nei bordi, senza scendere mai sotto un certo valore. Quest'intuizione ci permette di correggere la soluzione precedente inserendo come ultimo valore per le colonne delle tabelle da ridurre, una *stima* del valore successivo, ovvero la somma tra l'ultimo valore disponibile e la differenza tra quest'ultimo e il penultimo valore disponibile, ovvero l'aumento precedente, e per l'ultima riga delle tabelle una stima basata sulle colonne. Per l'ultimo valore nell'angolo in basso a destra invece, il ragionamento a righe o colonne fallisce e facciamo una *stima in diagonale* grazie all'ultimo elemento disponibile e il penultimo della diagonale. In formule quindi, indicando con *ncol* e *nrow* rispettivamente il numero di colonne e di righe della matrice originale, $ncol_{rid}$ il numero di colonne della matrice ridotta, e con $nrow_{rid}$ il numero di righe della matrice ridotta, con $actual_{row}$ la riga attuale, $actual_{col}$ la colonna attuale ed infine come al solito il fattore di riduzione con *reduction_factor* si ha:

$$left_{x_{ridotta}(ncol_{rid}-1,\ actual_{row})} =$$
$$left_{x_{originale}}(ncol-1,\ actual_{row} \times reduction_{factor}) +$$
$$abs(\ left_{x_{originale}(ncol-1,\ actual_{row} \times reduction_{factor})} -$$
$$left_{x_{originale}(ncol-2,\ actual_{row} \times reduction_{factor})}\ )$$

$$left_{x_{ridotta}(\ actual_{col},nrow_{rid}-1)} =$$
$$left_{x_{originale}}(\ actual_{col} \times reduction_{factor},\ nrow-1) +$$
$$abs(\ left_{x_{originale}(\ actual_{col} \times reduction_{factor},\ nrow-1)} -$$
$$left_{x_{originale}(\ actual_{col} \times reduction_{factor},\ nrow-2)}\ )$$

Infine, per l'ultimo elemento:

$$left_{x_{ridotta}(\ ncol_{rid}-1,nrow_{rid}-1)} =$$
$$left_{x_{originale}}(\ ncol-1,\ nrow-1) +$$
$$abs(\ left_{x_{originale}(\ ncol-1,\ nrow-1)} -$$
$$left_{x_{originale}(\ ncol-2,\ nrow-2)}\ )$$

In questo modo siamo riusciti a ridurre drasticamente gli effetti di bordo, ottenendo un RMSE in tali zone paragonabile a quello generale per una data configurazione. Riprendendo l'esempio di prima otteniamo ora:

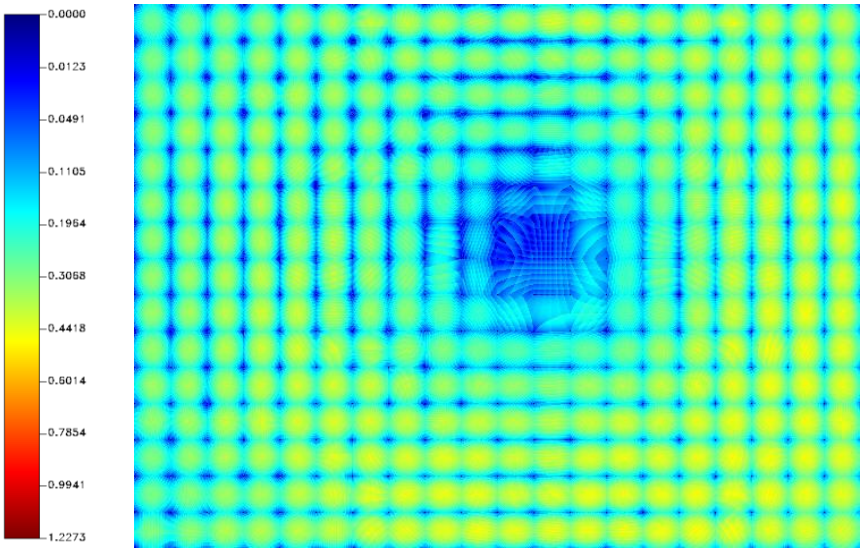


Figura 6.4.2.1 Effetti di bordo nell'immagine Left rettificata completamente rimossi dal secondo metodo, scala non lineare

Come è possibile notare gli effetti di bordo sono scomparsi del tutto. Notare che la configurazione non presenta un errore maggiore come invece potrebbe sembrare, ma è la scala a essere cambiata. Essa fa infatti riferimento sempre all'errore maggiore tra tutte le configurazioni disponibili, e siccome l'errore più alto si presentava sempre sui bordi in tutte le configurazioni, l'annullamento di quest'errore abbassa l'errore massimo e medio di ogni configurazione. Vediamo ora il risultato ottenuto dalla rettificazione tramite questa riduzione delle matrici, ricordando però che eventuali differenze non sono apprezzabili visivamente.

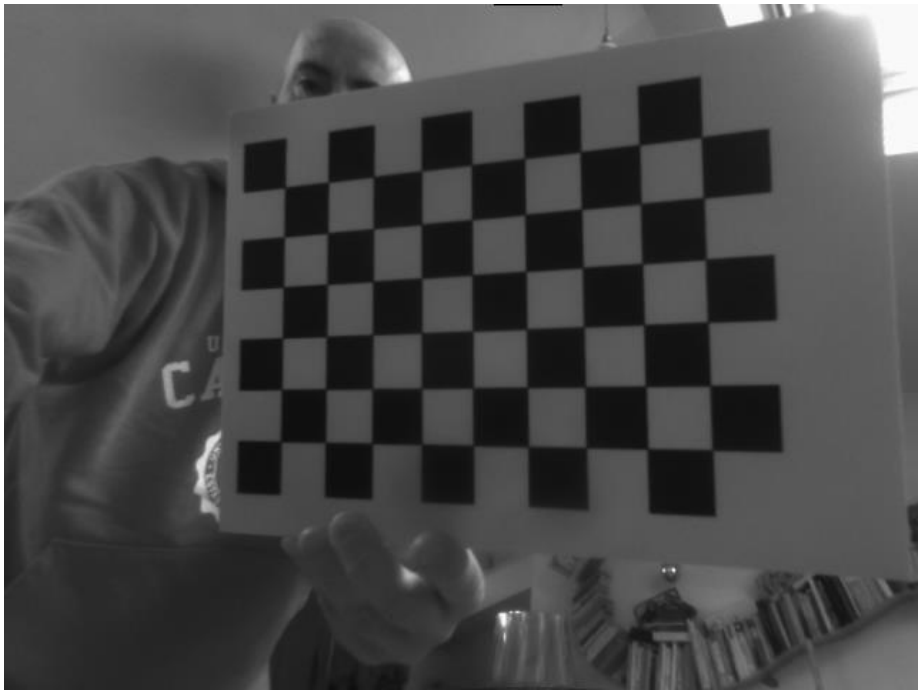


Figura 6.4.2.2 Effetti di bordo assenti nell'immagine Left rettificata con interpolazione bilineare, 4 bit per i fixed-point e reduction_factor 32 e metodo 2

Capitolo 7

Rettificazione adattiva

7.1 Distribuzione dell'errore nelle immagini

Grazie alle semplificazioni appena introdotte, siamo finalmente in grado di realizzare la rettificazione nei dispositivi *embedded*, proprio come prefissato, variando i parametri del numero di bit dedicati ai decimali per il fixed-point e il *reduction_factor* fino a ottenere il giusto compromesso tra fisica realizzabilità e qualità della rettificazione. Ci accorgiamo però che l'errore introdotto dalle semplificazioni non è disposto in modo uniforme lungo l'immagine, ma notiamo che l'errore è minore nel centro ottico dell'immagine e maggiore verso i bordi dell'immagine, come si vede, ad esempio, in figura *6.4.1.1* e *6.4.2.1*. Ciò è ovviamente dovuto alla natura della distorsione, che distorce l'immagine ai bordi molto più che al centro. Questo implica che non stiamo sfruttando a pieno le semplificazioni, in quanto alcune configurazioni che risulterebbero avere errore medio e massimo troppo alto per poter essere prese in considerazione, potrebbero avere aree con un errore più basso della media e dunque utilizzabile. Riuscire in questo compito, ovvero riuscire ad utilizzare diverse configurazioni per

diverse aree dell'immagine, ci permetterebbe di abbassare l'errore medio e massimo in alcuni punti critici.

7.2 Rettificazione adattiva

Il nuovo metodo che stiamo cercando, deve dunque permettere di avere le quattro matrici dei displacement ridotte in qualche modo con un *reduction_factor* diverso e magari un numero di bit per la parte decimale dei displacement in fixed-point diverso, per le diverse parti delle immagini, in modo da usare un *reduction_factor* maggiore e un numero di bit minore nelle aree dove l'errore è basso anche con questa configurazione, occupando meno memoria, e un *reduction_factor* minore e un numero di bit maggiore nelle aree dove l'errore è più alto, occupando più memoria, mantenendo dunque sempre un errore medio e massimo adeguato per tutta l'immagine. Vale la pena ricordare che l'errore, *l'RMSE,* in questione è quello riferito alle *tabelle dei displacement*, e cioè al mal posizionamento del punto tra i quattro pixel da interpolare nell'immagine, e non direttamente all'immagine risultante, a cui però è ovviamente legato, in quanto un mal posizionamento del punto farà sì che l'intensità per quel punto venga calcolata male, introducendo quindi un errore nell'immagine, per la quale definiamo un *RMSE* adeguato, basato sui pixel dell'immagine rettificata con float e i pixel dell'immagine rettificata con un metodo semplificato. L'idea elaborata per riuscire a effettuare questa operazione è quindi quella di dividere la matrice originale da ridurre in tante sottomatrici, e poi di ridurre con un certo fattore di riduzione e magari di utilizzare un numero adeguato di bit per i decimali del fixed-point ognuna di queste sottomatrici, in modo da ottenere un errore per

quest'area – o meglio sotto matrice – accettabile. Vediamo meglio con un esempio. Vogliamo suddividere la matrice originale, di dimensioni 640x480 - ovvero 370.200 × *sizeof(tipo_elementi_matrice)* bit - in tante sottomatrici di dimensioni NxM. Scegliendo Per esempio N = 32 e M = 32, si ha che la matrice originale viene dunque suddivisa in 300 sottomatrici da 32x32. Usiamo quindi una matrice di dimensione 20x15 - 640 / 32 = 20 e 480 / 32 = 15 - per contenere le 300 sottomatrici, ma così facendo non abbiamo ottenuto ancora nulla, che non sia una *diversa organizzazione* della matrice originale, che al momento occupa ancora 370.200 × *sizeof(tipo_elementi_matrice)* bit, cioè 32 × 32 × 300. Questo però ci permette di poter fare una cosa che prima non potevamo fare: ridurre *ogni sottomatrice,* le 300 da 32x32, con un fattore di riduzione diverso, mentre prima, infatti, l'unica opzione a disposizione era ridurre *tutta* la matrice originale di un certo *reduction_factor*. Come visto prima infatti, prendendo 16 ad esempio come *reduction_factor* per ridurre la matrice originale, otterremo una matrice ridotta di dimensioni 40x30, ed occuperà 1.200 × *sizeof(tipo_elementi_matrice)* bit. Essa, a parità di errore, rappresenta la minima dimensione occupabile tra tutti i metodi a disposizione, ma è anche molto inefficiente. Visto che in qualunque caso *campiona* sempre ogni *reduction_factor*, sia quando l'errore è basso, sia quando è alto. Con il nuovo metodo invece, sarebbe possibile occupare in ogni caso molta meno memoria rispetto alla tabella originale, ma non meno del metodo sopracitato, ma saremmo in grado di mantenendo un errore più basso dell'errore della matrice ridotta con il metodo precedente. Risparmieremo infatti memoria nelle zone in cui l'errore è basso e ne useremo di più in quelle in cui l'errore è alto, quindi da correggere al meglio. Possiamo quindi decidere di usare, per esempio, un *reduction_factor* pari a 32 per le sottomatrici 32x32 in cui l'errore è basso, ottenendo quindi delle sottomatrici ridotte di dimensioni 1x1, un *reduction_factor* di 16 per le sottomatrici in cui

l’errore è medio, ottenendo quindi delle sottomatrici ridotte di dimensioni 2x2, ed infine un *reduction_factor* di 8 per le sottomatrici in cui l’errore è alto, ottenendo quindi delle sottomatrici ridotte di dimensioni 4x4. In questo caso quindi non avremmo più 300 sottomatrici da 32x32, ma, visto che le abbiamo ridotte, 300 sottomatrici da 1x1, 2x2, 4x4, in numero vario, a seconda del numero di aree con errore basso, medio o alto.

Facciamo ora l’esempio in cui delle 300 sottomatrici, 50 siano ad errore basso, quindi delle 1x1, 180 siano ad errore medio, quindi delle 2x2, e le restanti 70 siano ad errore alto, quindi delle 4x4.

Avremmo quindi una dimensione totale di: 50 × 1×1 + 180 × 2×2 + 70 × 4×4 = 1.890 × *sizeof(tipo_elementi_matrice)* bit. Essa è di dimensioni maggiori della dimensione della matrice ridotta ottenuta riducendo tutta la matrice originale: 1.200 × *sizeof(tipo_elementi_matrice)* bit, ma rispetto a quest’ultima potrà sicuramente contare su una riduzione dell’errore maggiore dove l’errore è più alto. Chiariamo con delle immagini, per scoprire un problema secondario da risolvere. Prendiamo ad esempio la matrice dei displacement left_x.

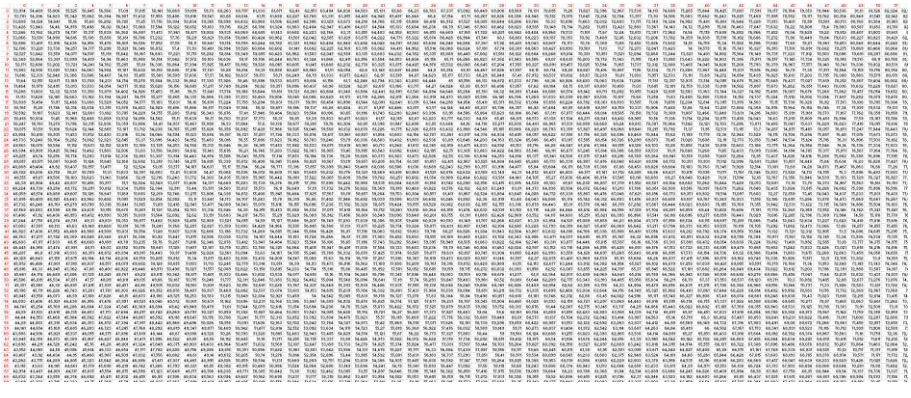


Figura 7.2.1 Matrice originale dei displacement left_x da ridurre, parzialmente visibile

Selezioniamo ora le 300 sottomatrici da 32x32.

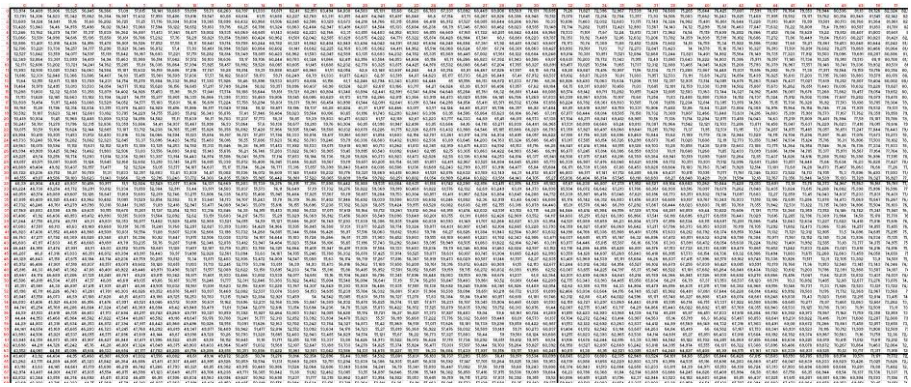


Figura 7.2.2 Selezione sottomatrici 32x32 dalla matrice originale dei displacement left_x da ridurre

Ora riduciamo per esempio ognuna di queste sottomatrici con un diverso *reduction_factor*, giusto a titolo di esempio. La prima, con indici (0, 0) nella matrice delle sottomatrici, sarà ridotta di un *reduction_factor* 16, ottenendo una 2x2, la seconda, con indici (1, 0), di un *reduction_factor* 32 ottenendo una 1x1, la terza, con indici (0, 1), di un *reduction_factor* 8 ottenendo una 4x4 e infine la quarta, con indici (1, 1), di un *reduction_factor* 8, ottenendo un'altra 4x4. Le sottomatrici scelte non sono casuali, dato che ci interessa proprio valutare cosa succede tra matrici adiacenti aventi diverso *reduction_factor*. Diamo uno sguardo alla matrice, 20x15, delle sottomatrici.

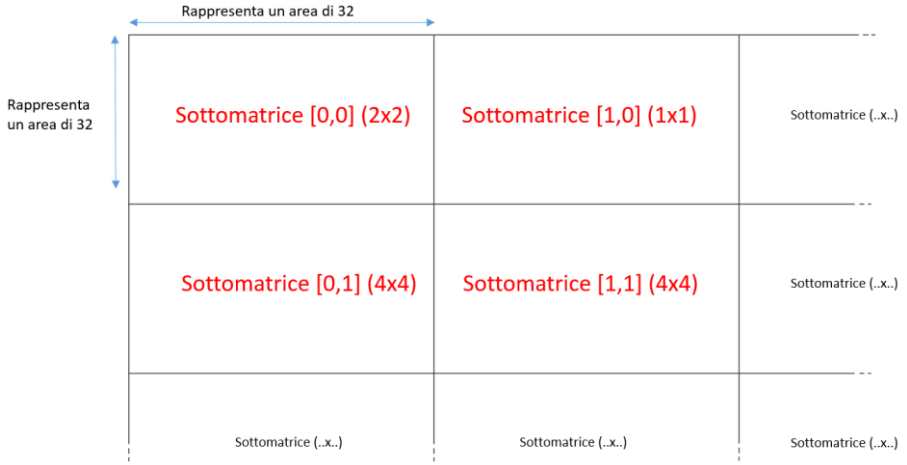


Figura 7.2.3 Sottomatrici dalla matrice delle sottomatrici, ridotte

Mantenendo, per semplificare, ancora valori float, avremo una situazione del genere:

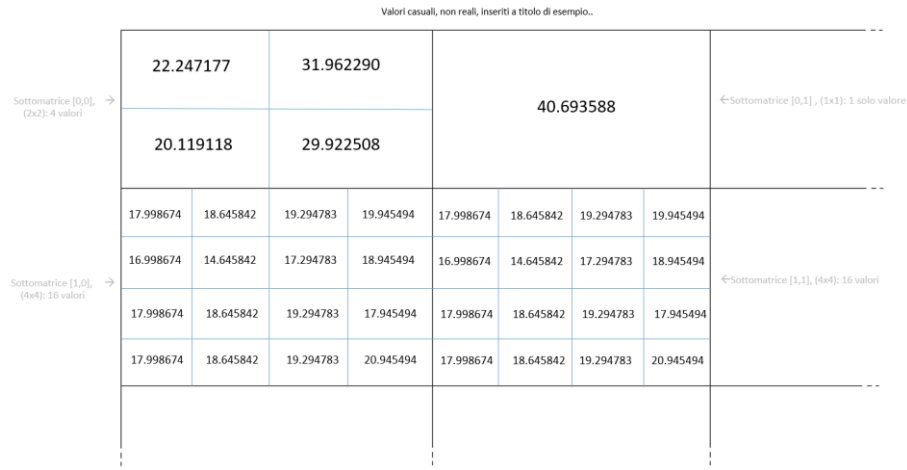


Figura 7.2.4 Dettagli sottomatrici ridotte dalla matrice delle sottomatrici

Notiamo a questo punto che se lasciassimo tutto com'è, avremmo seri problemi nell'estrazione dei displacement da questa matrice delle sottomatrici, e cioè quali sarebbero i 4 valori da andare ad interpolare.

Facciamo l'ipotesi in cui il pixel dell'immagine non distorta da interpolare sarebbe il pixel (12, 27), nell'estrarre il displacement andremmo a finire quindi nella zona indicata dal puntino rosso. Quali valori andrebbero presi?

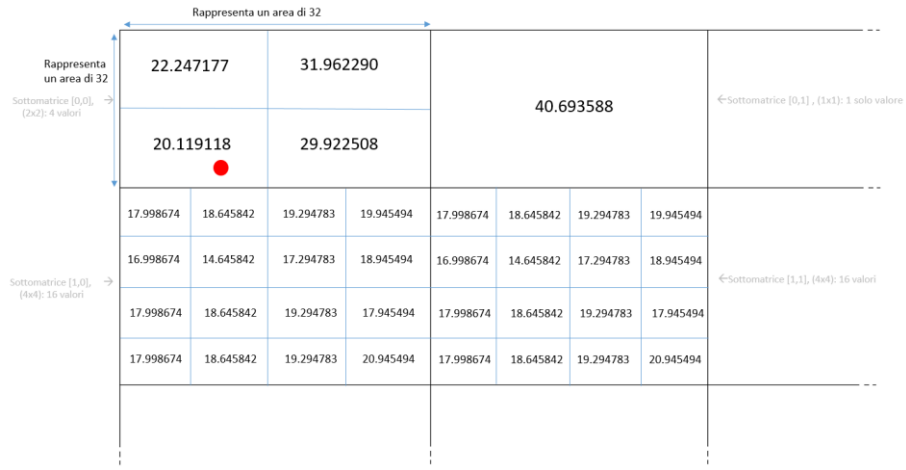


Figura 7.2.5 Quali displacement andrebbero presi?

Il problema dunque diventa complesso, visto che estrarre i displacement richiederebbe l'uso di una o più sottomatrici, e una complessità nel calcolo non adatto alle *FPGA*. Una soluzione esiste però, ed è anche abbastanza semplice: Aggiungere a ogni sottomatrice una riga e una colonna in più, che stavolta esistono a differenza del caso degli effetti di bordo, così da avere sempre i quattro displacement da estrarre nella stessa sottomatrice. I displacement aggiunti sono presi con lo stesso "passo", o *reduction_factor*, della sottomatrice, risolvendo anche il problema di disomogeneità tra sottomatrici vicine che potrebbe in alcuni casi addirittura peggiorare il risultato. Quest'aggiunta di "inutili" duplicati si rende necessaria per poter ridurre la complessità ad un livello tale da poter essere realmente mappato su *FPGA*. Vediamo dunque cosa accade con delle figure.

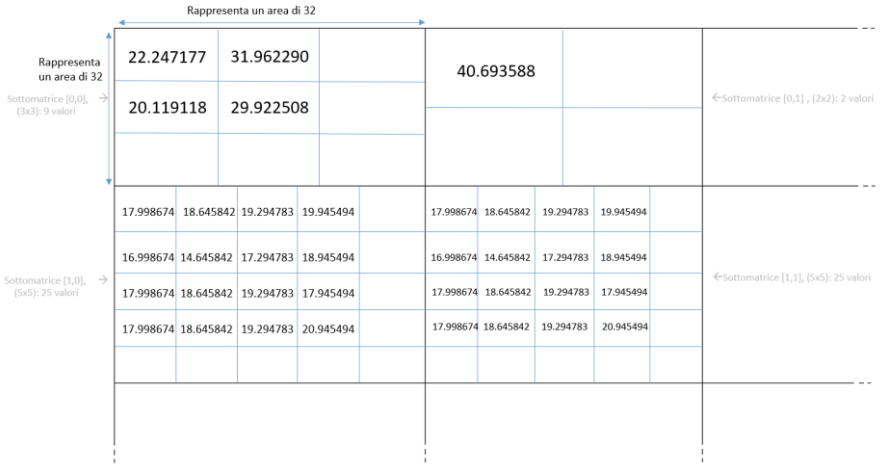


Figura 7.2.6 Aggiunta nuova riga e nuova colonna ad ogni sottomatrice

La soluzione funziona, poiché il punto ora potrebbe trovarsi solo all'interno delle vecchie aree, in rosso nell'immagine, e non nelle celle quelle aggiunte, dato che le sottomatrici, ora aumentate di una riga e una colonna, rappresentano sempre un'area di 32x32 per la matrice delle sottomatrici, e quindi se anche il punto fosse nella 33esima riga o colonna, andrebbe nella sottomatrice adatta.

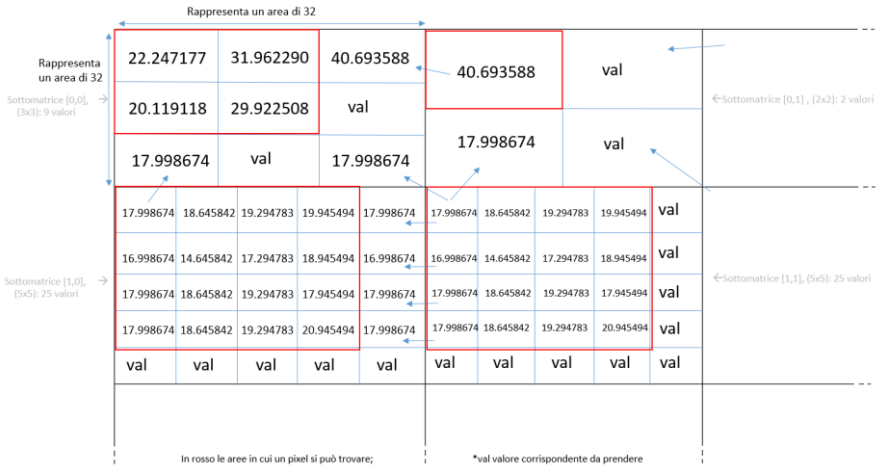


Figura 7.2.7 Valori duplicati e nuovi valori campionati aggiungendo una riga e una colonna in più. In rosso le aree in cui un punto può effettivamente capitare

A questo punto l'estrazione dei displacement diventa semplice, basta solo una sottomatrice e passare da un sistema di indici *assoluto* nella matrice originale o ridotta, ad uno *relativo* che fa capo ad uno assoluto. Ovvero prima si individua la sottomatrice giusta tramite l'individuazione del *matrix_index_y* e *matrix_index_x* tramite *divisione tra interi* delle coordinate attuali del pixel, di cui vogliamo interpolare l'intensità, e il valore *N* scelto per il *matrix_index_x,* e quello *M* per il *matrix_index_y*. A questo punto sottraiamo alle coordinate x e y rispettivamente l'indice appena trovato moltiplicato N per le x ed M per le y, ottenendo così le coordinate del punto cercato nella sottomatrice. A quel punto estraiamo le coordinate del punto nell'immagine distorta come al solito tramite interpolazione bilineare.

Per poter realizzare realmente le cose non abbiamo bisogno però di avere una semplice matrice di matrici, ma una matrice di strutture, in cui ogni struttura è composta dalla sottomatrice, il suo fattore di riduzione (int), e nel caso volessimo variare anche il fixed-point nelle varie sottomatrici, il numero di bit da usare per la parte decimale (int), ovvero il numero di shift da fare a destra o sinistra, oppure direttamente il suo equivalente 2^{nbit}. Rifacciamo ora i calcoli riprendendo l'esempio di iniziale con la nuova soluzione, avremmo quindi una dimensione totale di:

50 × 2x2 + 180 × 3x3 + 70 × 5x5 = 3.570 × *sizeof(tipo_elementi_matrice)* bit

A cui dobbiamo aggiungere:

300 × *sizeof(int_per_rappresentare_reduction_factof)* bit

E nel caso volessimo variare anche i fixed-point:

300 × *sizeof(int_per_rappresentare_bit_fixed_point)* bit

Paragonata alla dimensione occupata della matrice ridotta per intero resta comunque maggiore, e maggiore anche rispetto all'iniziale calcolo del caso ideale, ma resta comunque un quantitativo molto basso, tale da poter essere inserito nelle *FPGA*, e rappresenta un buon compromesso per ottenere una qualità sicuramene migliore. Vediamo infine un esempio di rettificazione ottenuta con questo novo metodo.

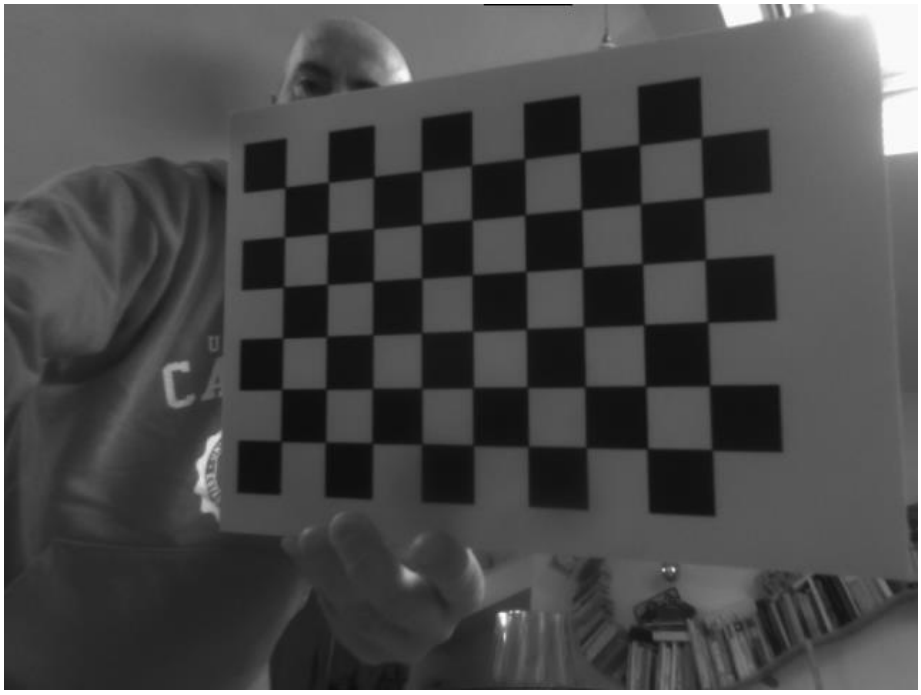


Figura 7.2.8 Immagine rettificata con la nuova rettificazione adattiva

Come vedremo nel capito successivo, al fine di ridurre al minimo le dimensioni occupate dalle tabelle ridotte con il nuovo metodo, scegliamo N ed M come la più grande potenza di 2 in grado di dividere in numero intero rispettivamente la *width* e la *height* della tabella da ridurre. Così facendo otteniamo però aree più grandi e quindi potenzialmente più inclini ad avere un errore eterogeneo, tuttavia

ridurremo sicuramente la dimensione finale occupata, in quanto per ogni sottomatrice in più si è costretti ad aggiungere una riga e una colonna in più.

Capitolo 8

Risultati sperimentali

8.1 Immagini di riferimento wide angle

Come in precedenza, mostreremo ora le immagini di riferimento, da rettificare, utilizzate nei test a seguire.

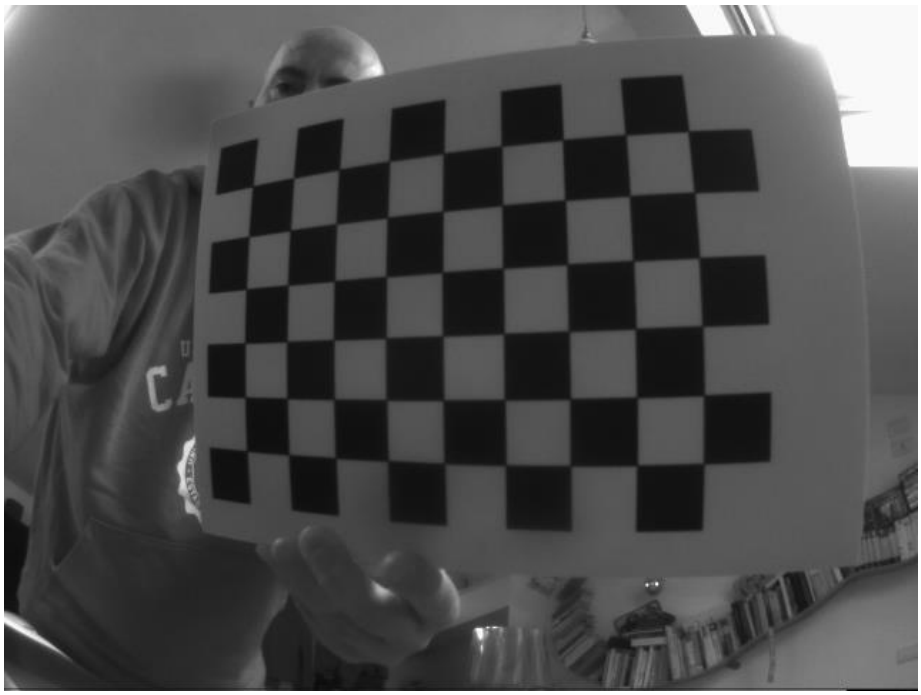


Figura 8.1.1 Immagine left di riferimento wide angle da rettificare

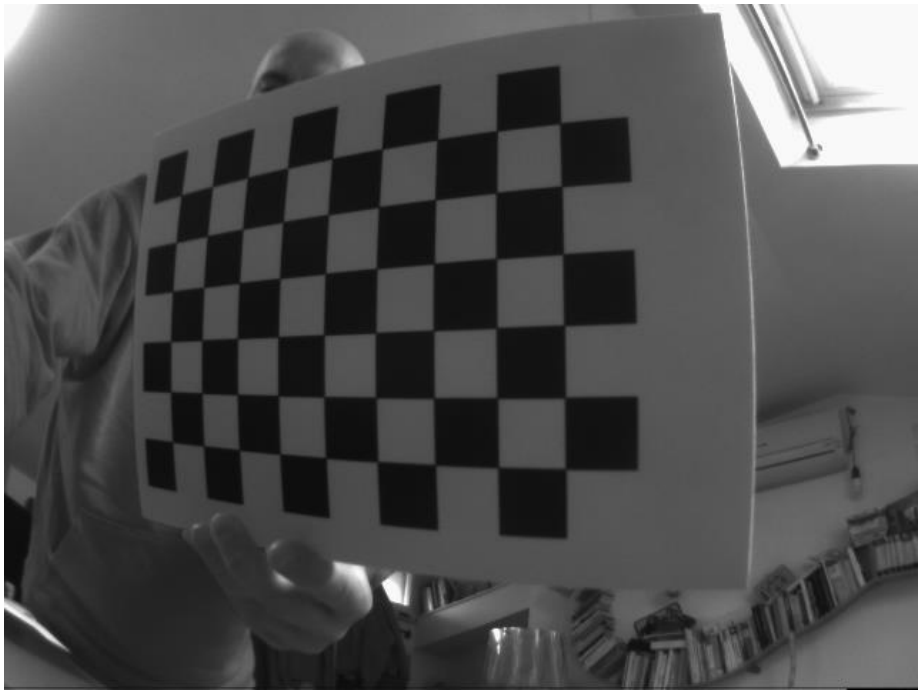


Figura 8.1.2 Immagine left di riferimento wide angle da rettificare

Il *dataset* in questione è dunque quello con lenti wide angle. Ribadiamo però, che mostreremo entrambe le immagini ottenute solamente nel caso in cui dovesse rendersi necessario, data la similitudine dei risultati.

8.2 Rettificazione senza interpolazione

Per questo tipo di rettificazione, con errori molto marcati, ci limitiamo a mostrare solo il risultato.

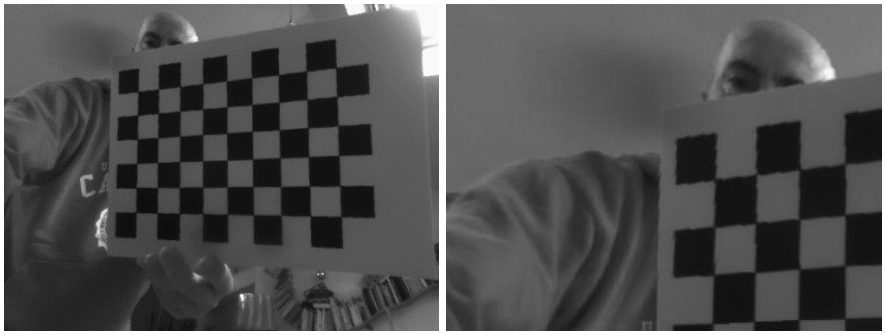


Figura 8.2.1 Immagine left rettificata senza interpolazione, artefatti ed errori molto evidenti

8.3 Rettificazione con float

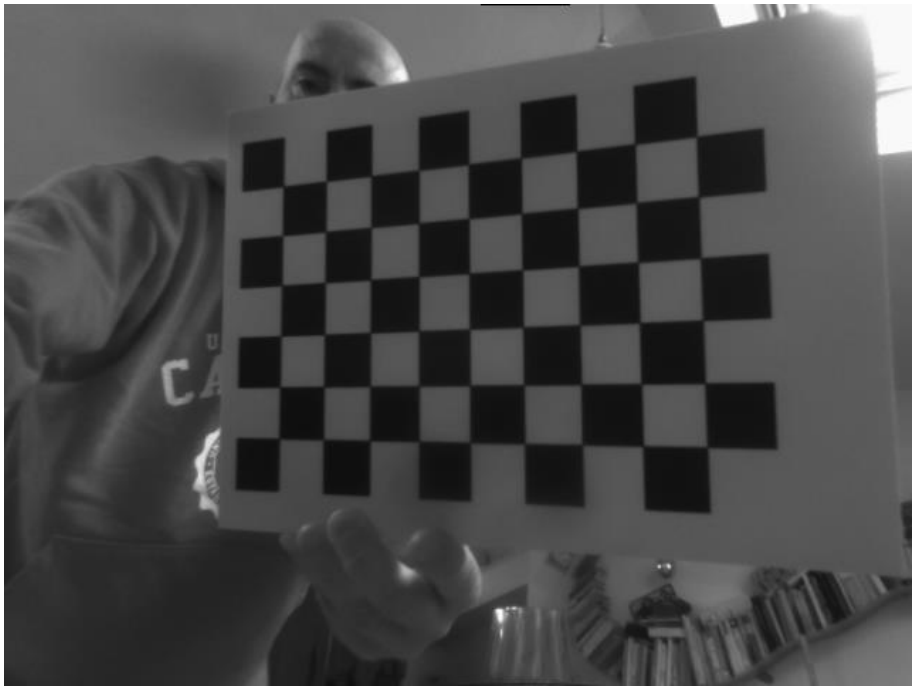


Figura 8.3.1 Immagine left rettificata con float.

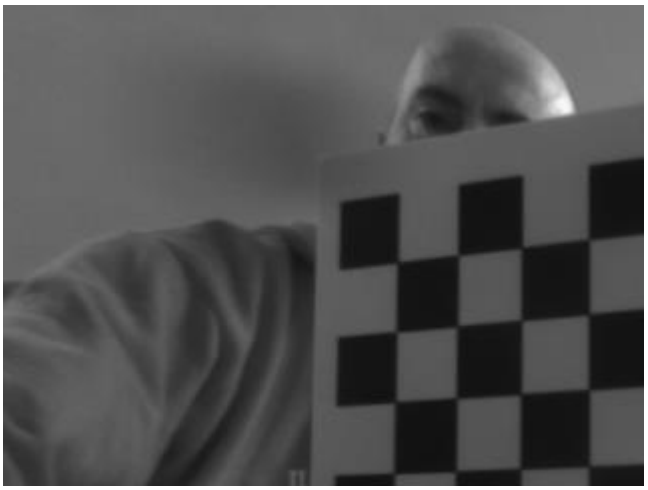


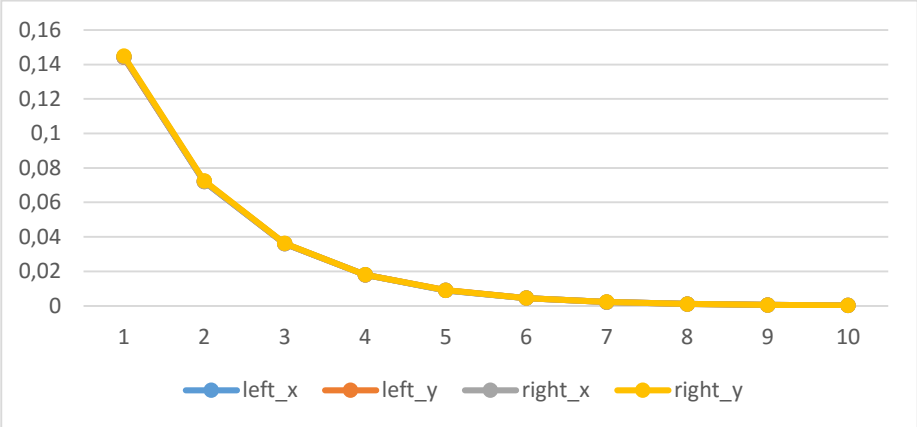
Figura 8.3.2 Dettagli immagine left rettificata con float ed interpolazione bilineare.

Come già anticipato, quest'immagine rappresenta il risultato da raggiungere dai metodi semplificati descritti in precedenza, e a cui faremo sempre riferimento.

8.4 Rettificazione con fixed-point

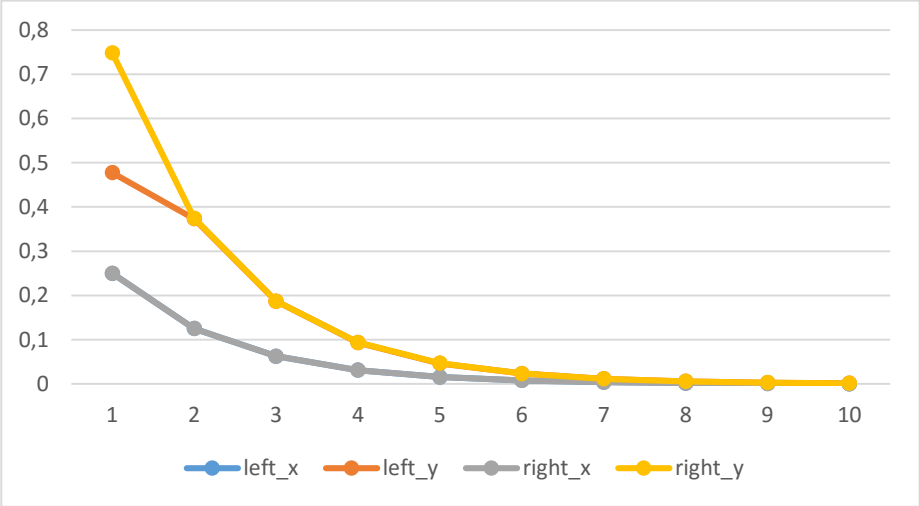
Per questo di tipo di semplificazione l'unica variabile in gioco è il numero di bit per la parte decimale del fixed-point. Variando questo parametro infatti, si è visto che l'*RMSE medio* e *massimo,* sia riferito alle singole tabelle *x* e *y,* sia riferito ad entrambe, per ogni bit in più, a partire dalla configurazione con errore maggiore, ovvero quella con un solo bit per la parte decimale, dimezza la sua entità, fino ad arrivare a valori tendenti allo 0. Si è visto dunque, che per il sistema utilizzato, si ottiene un errore accettabile già a partire dall'utilizzo di 4 bit, per arrivare ad un livello molto buono utilizzandone almeno 6.

L’errore dopo questo valore è piuttosto basso e incrementare ulteriormente il numero di bit non comporta grossi benefici come in precedenza.



nbit fixed-point				
1	*0,144462*	*0,144154*	*0,144407*	*0,145015*
2	*0,072154*	*0,072177*	*0,072126*	*0,072598*
3	*0,036083*	*0,036115*	*0,036088*	*0,036286*
4	*0,01806*	*0,018038*	*0,01803*	*0,018132*
5	*0,009016*	*0,009042*	*0,009027*	*0,009079*
6	*0,004511*	*0,004509*	*0,004511*	*0,004538*
7	*0,002254*	*0,002255*	*0,002257*	*0,002268*
8	*0,001128*	*0,001129*	*0,001127*	*0,001135*
9	*0,000564*	*0,000565*	*0,000564*	*0,000567*
10	*0,000283*	*0,000282*	*0,000282*	*0,000284*
	left_x	*left_y*	*right_x*	*right_y*

Figura 8.4.1 RMSE medio singole matrici



nbit fixed-point				
1	0,25	0,477362	0,25	0,748492
2	0,125	0,373454	0,125	0,374562
3	0,0625	0,186889	0,0625	0,187097
4	0,03125	0,093443	0,03125	0,093733
5	0,015625	0,046423	0,015625	0,046838
6	0,007813	0,023353	0,007813	0,023429
7	0,003906	0,011633	0,003906	0,011671
8	0,001953	0,005856	0,001953	0,005857
9	0,000977	0,002797	0,000977	0,002926
10	0,000488	0,001456	0,000488	0,001465
	left_x	left_y	right_x	right_y

Figura 8.4.2 RMSE massimo singole matrici.

Dati simili si ottengono per l'*RMSE* di entrambe le matrici, che valutiamo questa volta tramite l'utilizzo di *colormap*, applicate in questo caso all'*RMSE*, identifichiamo *visivamente* l'errore:

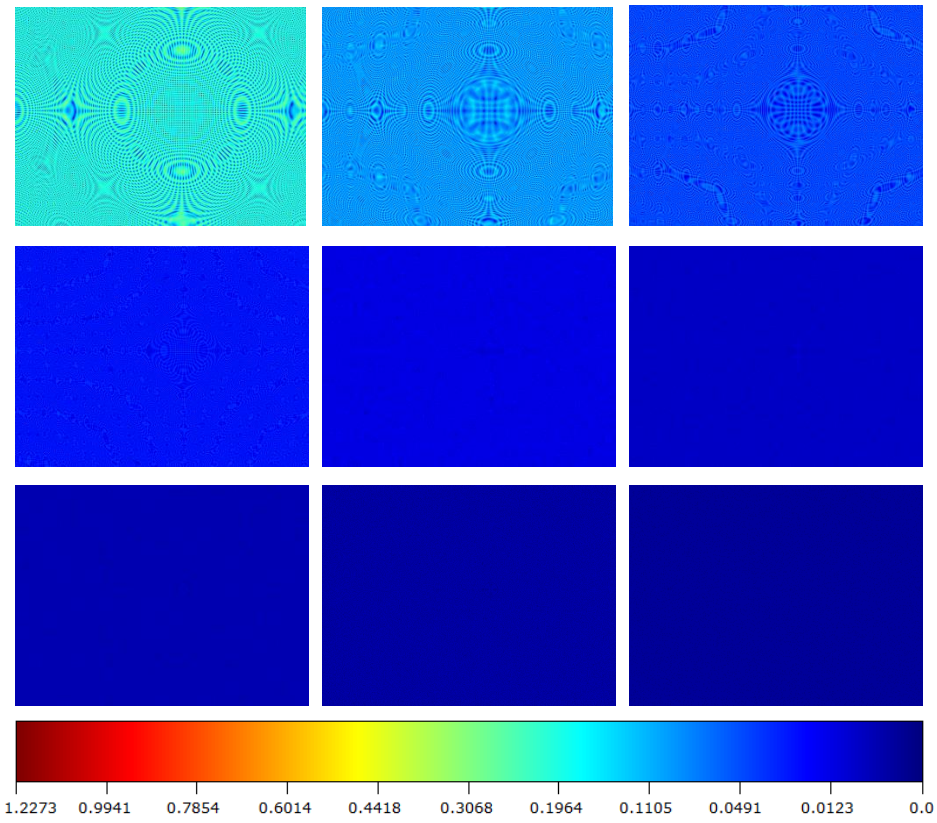


Figura 8.4.3 RMSE di entrambe le matrici x e y, fixed-point da sinistra a destra da 1 a 9 bit.

Vediamo ora le immagini generate da questa rettificazione:

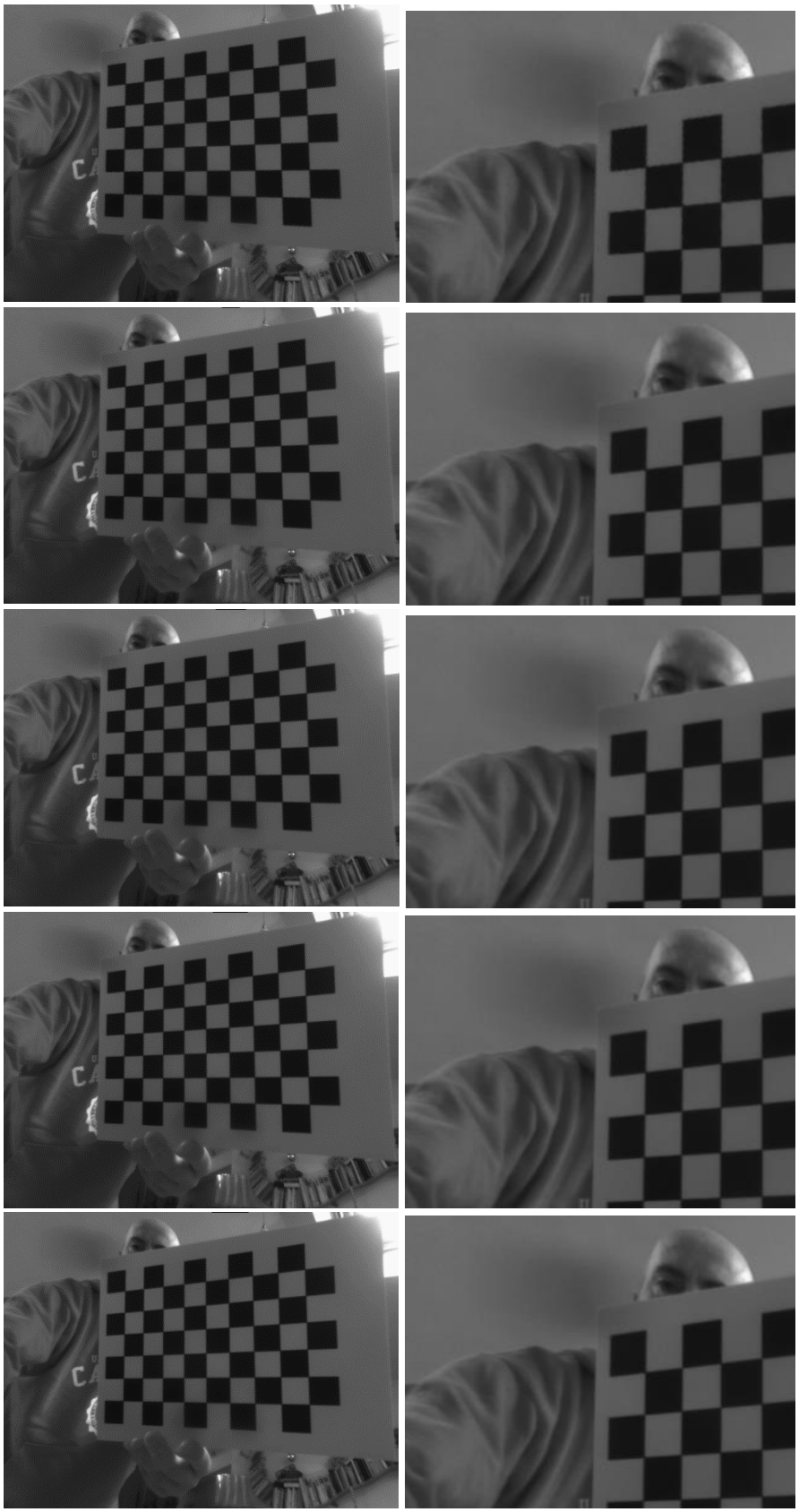


Figura 8.4.4 Immagine left rettificata con fixed-point da 1 a 5

Ovviamente ci limitiamo a mostrare pochi risultati, in quanto visivamente parlando, non si notano differenza sostanziali rispetto ad una rettificazione float usando un numero di bit maggiore di 4. Vediamo infine le colormap riferite *all'RMSE* dei pixel dell'immagine.

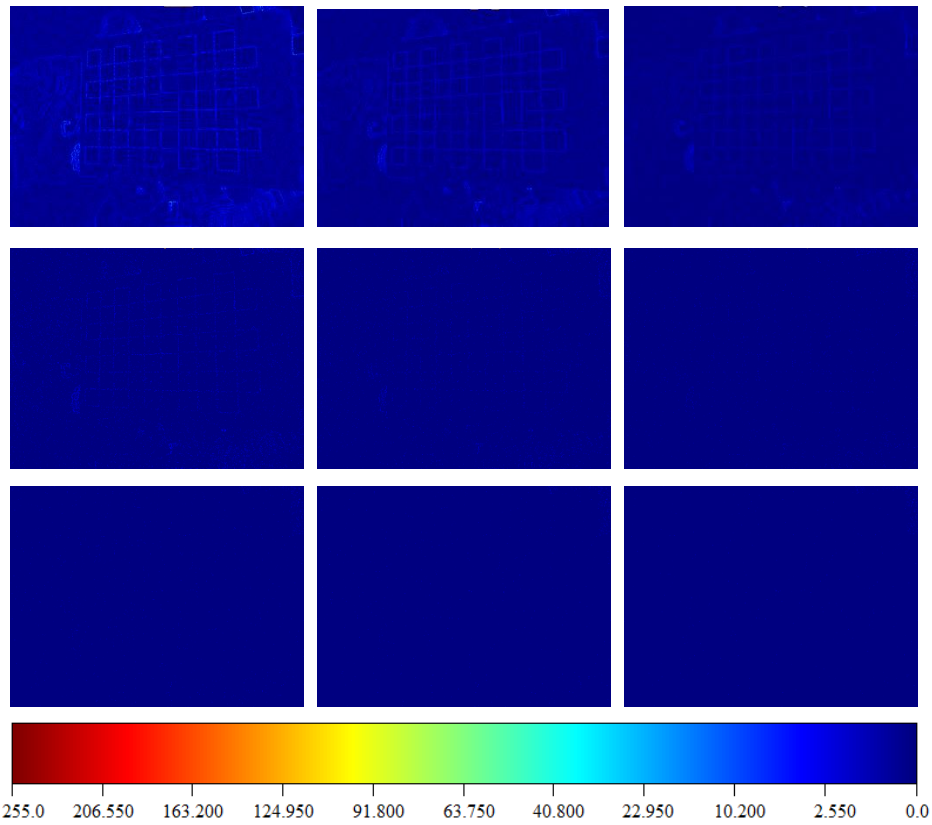
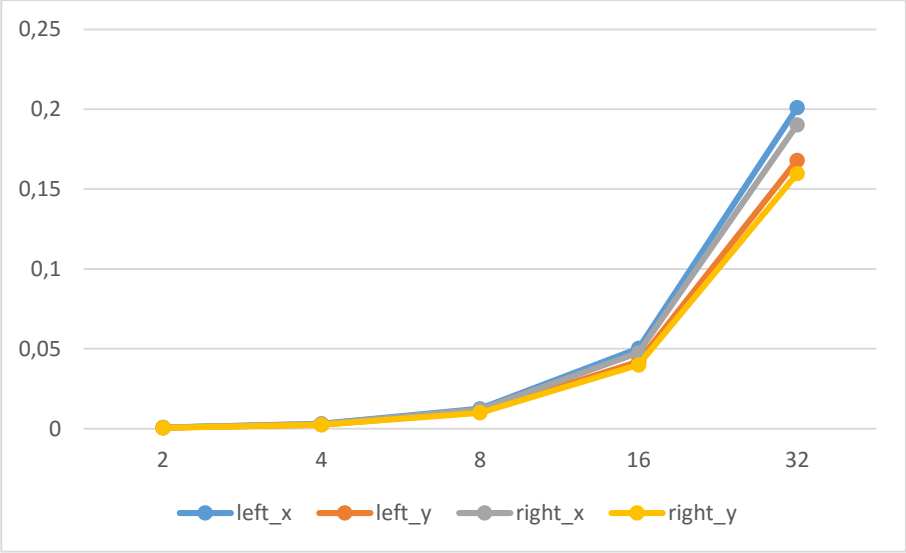


Figura 8.4.5 RMSE pixel immagine, fixed-point, da sinistra a destra da 1 a 9 bit.

Notiamo infatti che l'errore, anche se presente in molti punti dell'immagine, è constatabile nei contorni degli oggetti molto più che altrove. Ciò avviene a causa del fatto che l'interpolazione è migliore nel momento in cui i quattro pixel da interpolare variano di poco tra loro nell'intensità. Il mal posizionamento del punto di cui vogliamo interpolare l'intensità in queste zone è dunque critico.

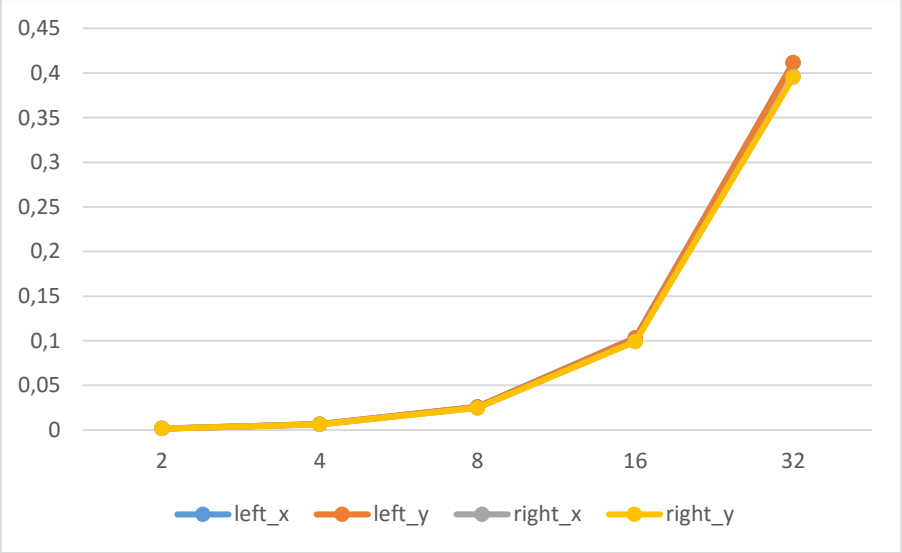
8.5 Rettificazione con matrici ridotte e float

Questa semplificazione, anche se non implementabile nei dispositivi *embedded*, viene valutata per quantificare l'errore dovuto alla singola riduzione delle matrici, e come non come vedremo più avanti l'effetto congiunto di quest'ultimo e del fixed-point. L'unica variabile in gioco qui è *reduction_factor*. Variando questo parametro infatti, si è visto che l'*RMSE medio* e *massimo*, sia riferito alle singole tabelle *x* e *y*, sia riferito ad entrambe, per ogni potenza di 2 utilizzata, fino a 32, presenta un andamento di tipo esponenziale, la quale, partendo da valori prossimi allo zero, quadruplica circa la sua entità, fino ad arrivare a valori molto alti utilizzando 32 come *reduction_factor*. Si è visto dunque, che per il sistema utilizzato, si ottiene un errore accettabile utilizzando al massimo 16 come *reduction_factor*. L'errore medio, ma soprattutto massimo oltre questa configurazione è troppo alto per poter pensare di utilizzare tali configurazioni.



reduction_factor				
2	0,000707	0,000595	0,000669	0,000566
4	0,00307	0,002571	0,002906	0,002443
8	0,012496	0,010452	0,011833	0,009932
16	0,050204	0,041983	0,047543	0,039894
32	0,200935	0,167985	0,190283	0,159627
	left_x	left_y	right_x	right_y

Figura 8.5.1 RMSE medio singole matrici



reduction_factor				
2	0,00165	0,00165	0,00159	0,00159
4	0,00653	0,0065	0,00623	0,00623
8	0,02588	0,02585	0,02484	0,02484
16	0,10328	0,10306	0,09937	0,09909
32	0,41142	0,41165	0,39639	0,3952
	left_x	left_y	right_x	right_y

Figura 8.5.2 RMSE massimo singole matrici.

Dati simili si ottengono per l'*RMSE* di entrambe le matrici, che valutiamo anche questa volta tramite l'utilizzo di colormap, applicate in questo caso all'*RMSE*, identifichiamo *visivamente* l'errore:

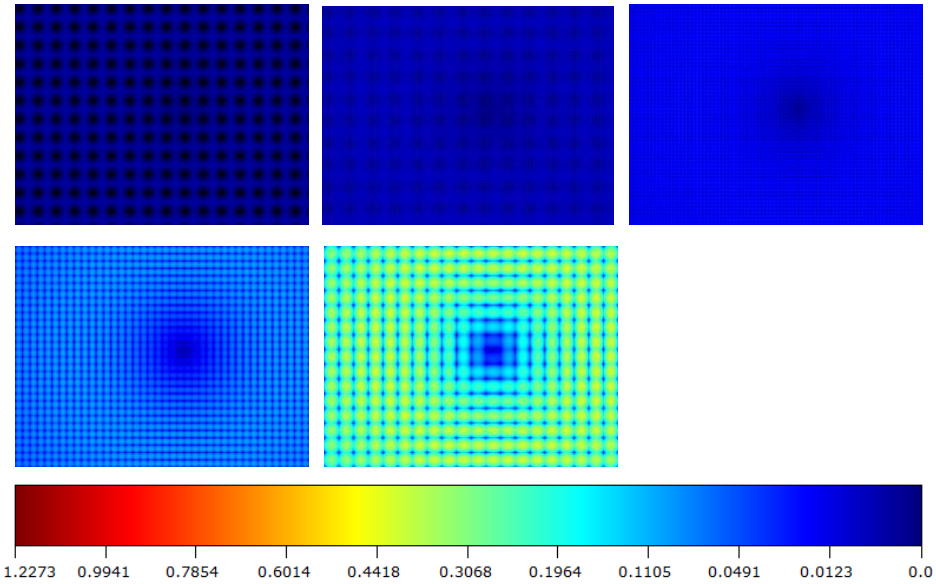


Figura 8.5.3 RMSE di entrambe le matrici x e y, reduction_factor da sinistra a destra da 2 a 32.

I punti neri nelle immagini originali, qui accorpati in qualche modo, rappresentano i punti che sono sicuramente ad errore prossimo allo 0, in quanto il displacement campionato esiste e quindi l'interpolazione per quei punti equivale ad utilizzare quel displacement. La rettificazione con questo metodo non presenta artefatti nettamente visibili, e quindi, da questo momento in poi, non mostreremo più le immagini rettificate, dato che conosciamo già gli effetti del fixed-point e anche quando uniremo i due fattori, a meno di un'analisi pixel per pixel, sarebbe inutile mostrarle. Ci concentreremo invece maggiormente sull'errore. Per questo metodo è interessante anche notare le colormap delle singole matrici, che ci indica dove è localizzato maggiormente l'errore all'interno di quest'ultime. Notiamo infatti che l'errore è maggiormente presente nei bordi, ma esistono zone anche in quelle con *reduction_factor* 32 con errore molto basso.

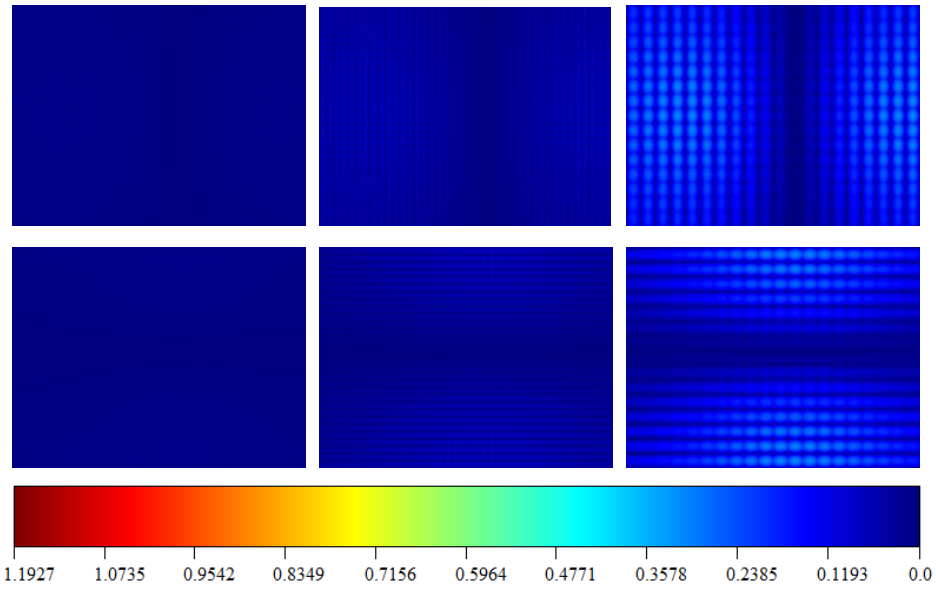


Figura 8.5.4 Colormap singole matrici, rispettivamente in alto matrice left_x, e in basso matrice left_y, reduction_factor da 8 a 32

Vediamo infine le colormap riferite *all'RMSE* dei pixel dell'immagine.

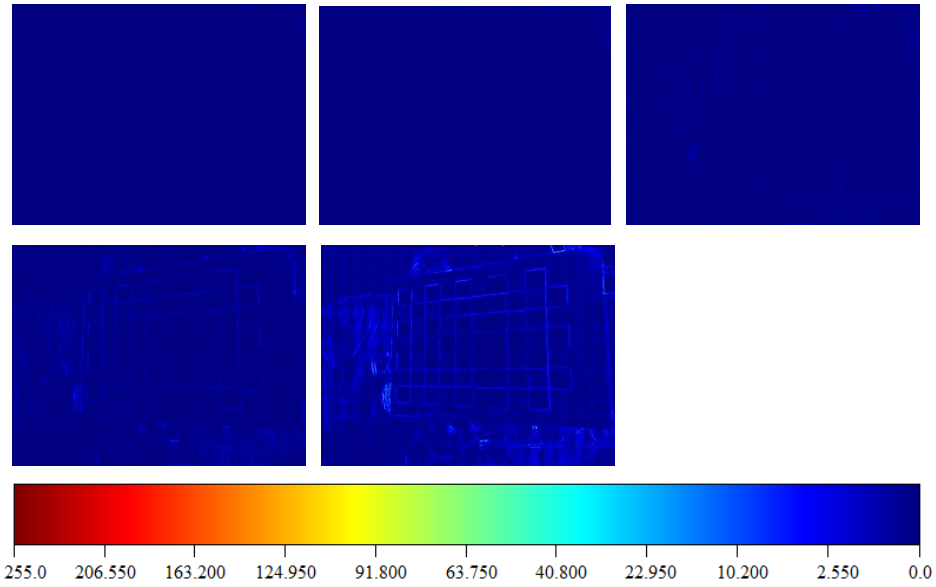
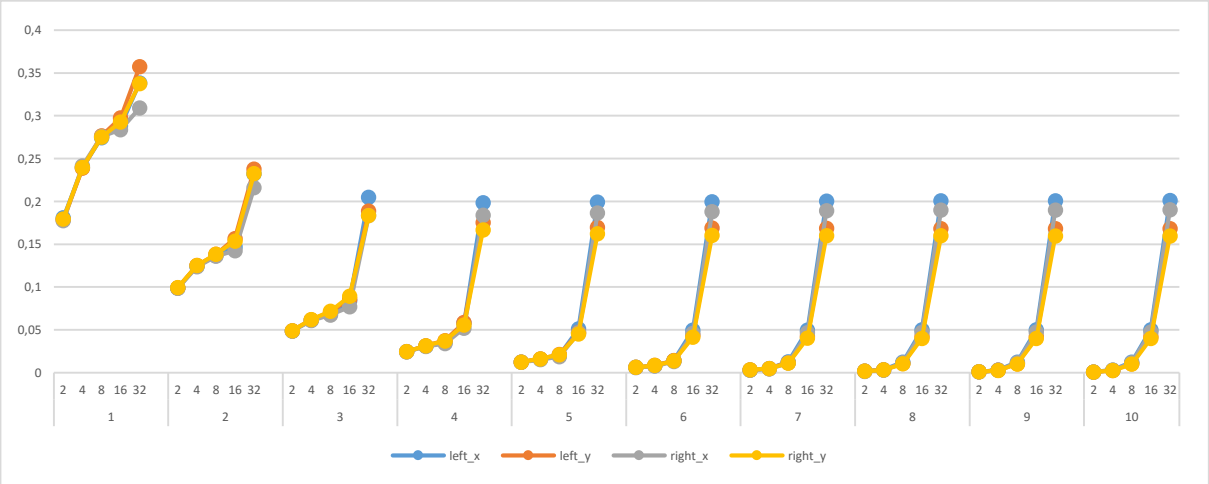


Figura 8.5.5 RMSE pixel immagine, da sinistra a destra da 2 a 32.

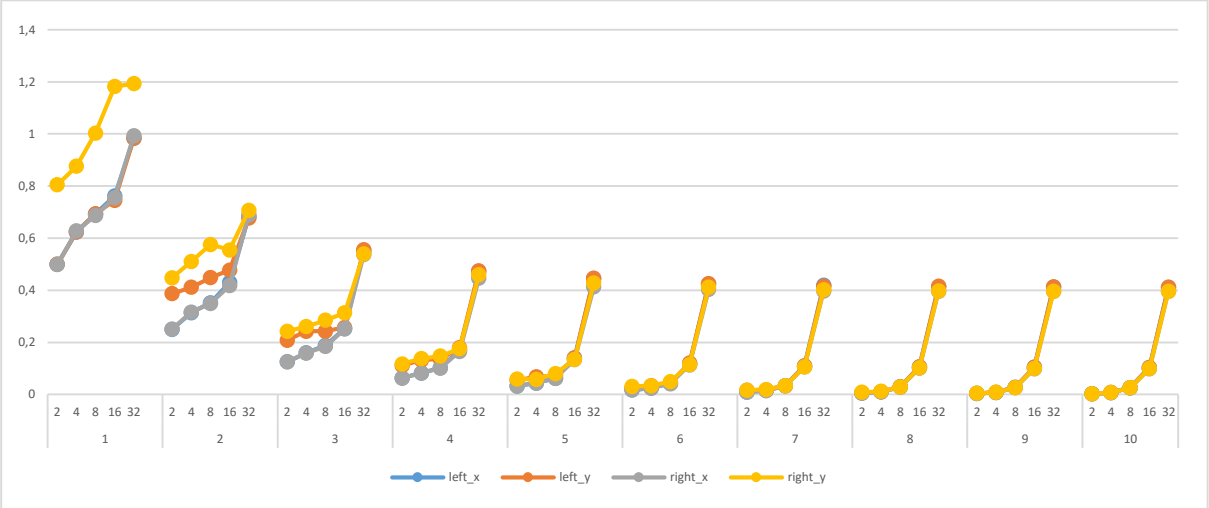
8.6 Rettificazione con matrici ridotte e fixed-point

In questa semplificazione valutiamo finalmente l'errore congiunto dovuta all'utilizzo del fixed-point e delle matrici ridotte. Le variabili in gioco sono dunque due: il *reduction_factor* e il numero di bit per la parte decimale del fixed-point. Utilizzando le stesse configurazioni di prima infatti, *nbit 1 – 10* e *reduction_factor* 2 – 32, otteniamo 50 possibili configurazioni, che però non mostreremo nella loro totalità. Dall'analisi dei dati è dunque emerso che il fattore predominante che caratterizza *l'RMSE*, sia *massimo* che *medio*, e sia delle singole matrici che di entrambe le matrici è il *reduction_factor*. Variando infatti il fixed-point e mantenendo il *reduction_factor* fisso, si ottiene lo stesso effetto già notato in precedenza, ovvero l'errore diminuisce molto velocemente fino ai 4 bit, mentre tale variazione non è così veloce dopo questa soglia, arrivando infine, ad un punto in cui non ha praticamente alcun effetto incrementare tale valore.



nbit fixed-pointt	reduction_factor				
	2	0,18075	0,17862	0,17765	0,17945
	4	0,23945	0,23893	0,24146	0,23975
1	8	0,27438	0,27647	0,27578	0,27498
	16	0,28804	0,29737	0,28375	0,29266
	32	0,09874	0,09905	0,09876	0,09956
	2	0,12459	0,12505	0,12359	0,12502
	4	0,13661	0,1382	0,1359	0,13841
2	8	0,1469	0,15666	0,14246	0,15366
	16	0,23198	0,23753	0,21587	0,23229
	32	0,04872	0,04896	0,04865	0,04906
	2	0,06136	0,06167	0,06081	0,06209
	4	0,06839	0,07072	0,06728	0,07155
3	8	0,08423	0,08597	0,07693	0,08918
	16	0,20479	0,18857	0,18336	0,18338
	32	0,02451	0,02453	0,02436	0,02461
	2	0,03101	0,03147	0,03065	0,03153
	4	0,03563	0,03736	0,03389	0,03706
4	8	0,05788	0,05859	0,05194	0,05567
	16	0,19861	0,17514	0,18391	0,16658
	32	0,0123	0,01239	0,01226	0,01244
	2	0,01559	0,01612	0,01525	0,01612
	4	0,02024	0,02122	0,01864	0,02088
5	8	0,05103	0,04691	0,04656	0,04513
	16	0,19938	0,16971	0,18642	0,16222
	32	0,00612	0,00623	0,00607	0,00625
	2	0,00808	0,00847	0,00775	0,00847
	4	0,01431	0,01407	0,01293	0,0137
6	8	0,04972	0,0436	0,04597	0,04156
	16	0,19969	0,16907	0,18781	0,16047
	32	0,0031	0,0032	0,00303	0,0032
	2	0,00475	0,00492	0,00435	0,00484
	4	0,01275	0,01164	0,01148	0,01112
7	8	0,0497	0,04252	0,04649	0,04042
	16	0,20028	0,16837	0,18903	0,15993
	32	0,00165	0,00171	0,00156	0,00169
	2	0,00347	0,00334	0,00311	0,00324
	4	0,01239	0,01078	0,01144	0,01029
8	8	0,04998	0,04215	0,04696	0,04011
	16	0,20062	0,1682	0,1897	0,15982
	32	0,001	0,00101	0,00092	0,00099
	2	0,00311	0,0028	0,00282	0,00269
	4	0,01239	0,01057	0,01159	0,01006
9	8	0,05005	0,04209	0,04726	0,03998
	16	0,20078	0,16807	0,18997	0,15973
	32	0,00076	0,00072	0,0007	0,0007
	2	0,00304	0,00264	0,00282	0,00252
	4	0,01244	0,0105	0,01169	0,00998
10	8	0,05014	0,04203	0,04739	0,03994
	16	0,20088	0,16803	0,19012	0,1597
	32	0,41142	0,41165	0,39639	0,3952
		left_x	left_y	right_x	right_y

Figura 8.6.1 RMSE medio singole matrici



nbit fixed-point	reduction_factor				
1	2	0,49908	0,50015	0,49957	0,80484
	4	0,62387	0,62506	0,62805	0,87621
	8	0,69348	0,69305	0,68689	1,00342
	16	0,76148	0,74539	0,75482	1,18184
	32	0,9837	0,98413	0,99347	1,19273
2	2	0,24982	0,38704	0,25131	0,4476
	4	0,31421	0,41135	0,31665	0,51085
	8	0,35187	0,44858	0,34949	0,57585
	16	0,43018	0,47736	0,41791	0,55484
	32	0,68201	0,67731	0,68927	0,70658
3	2	0,12555	0,20859	0,12524	0,24266
	4	0,15961	0,24254	0,15949	0,26085
	8	0,18878	0,24254	0,18463	0,28523
	16	0,25238	0,25772	0,25183	0,31334
	32	0,55066	0,55631	0,53693	0,53973
4	2	0,06366	0,11077	0,0636	0,11766
	4	0,08234	0,1319	0,08188	0,13774
	8	0,10248	0,13608	0,1012	0,1473
	16	0,1709	0,17966	0,16547	0,17596
	32	0,47235	0,47406	0,44623	0,46014
5	2	0,03247	0,05504	0,03278	0,05877
	4	0,04376	0,06781	0,04352	0,05841
	8	0,0622	0,07257	0,06213	0,08081
	16	0,14087	0,14002	0,13373	0,13373
	32	0,44464	0,44653	0,41266	0,4281
6	2	0,01709	0,02671	0,01703	0,03076
	4	0,02454	0,03309	0,02417	0,03411
	8	0,04395	0,04346	0,04108	0,04921
	16	0,11786	0,12021	0,11346	0,11508
	32	0,42377	0,42523	0,40332	0,41187
7	2	0,00934	0,01577	0,00928	0,01683
	4	0,01495	0,01777	0,01489	0,01892
	8	0,03388	0,03394	0,03357	0,0336
	16	0,11023	0,10965	0,10632	0,10602
	32	0,41907	0,41528	0,39746	0,40158
8	2	0,00549	0,00713	0,00537	0,00865
	4	0,0105	0,01065	0,01019	0,01228
	8	0,02972	0,02982	0,02795	0,02884
	16	0,10632	0,10587	0,10095	0,10193
	32	0,41339	0,41504	0,39655	0,39768
9	2	0,00354	0,0038	0,00348	0,00525
	4	0,00836	0,00827	0,008	0,00886
	8	0,02759	0,02762	0,02606	0,02643
	16	0,10474	0,10477	0,09906	0,10031
	32	0,41144	0,41309	0,39655	0,39572
10	2	0,00256	0,00259	0,0025	0,00342
	4	0,00739	0,00736	0,00708	0,00745
	8	0,02649	0,02667	0,02539	0,02564
	16	0,10376	0,10391	0,09877	0,09979
	32	0,41095	0,41184	0,39655	0,39563
		left_x	left_y	right_x	right_y

Figura 8.6.2 RMSE massimo singole matrici.

Inseriamo ora un altro grafico interessante, che utilizza gli stessi dati ma in modo da variare il fixed-point prima di variare il *reduction_factor*, il quale mostra al meglio il compromesso come l'errore con *reduction_factor* 32 si discosti molto dalle altre configurazioni.

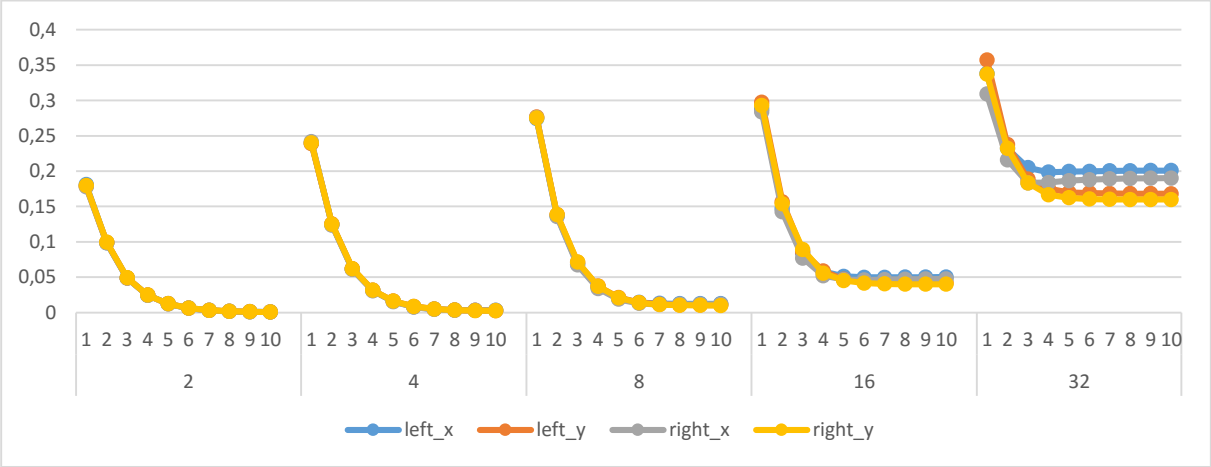


Figura 8.6.1.1 RMSE medio singole matrici

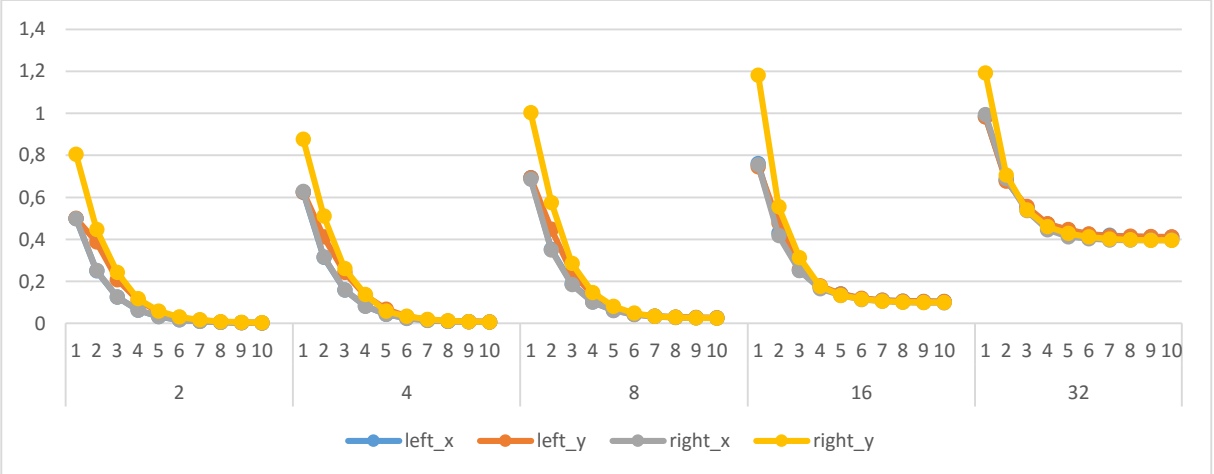


Figura 8.6.2.1 RMSE massimo singole matrici

Dati simili si ottengono per l'*RMSE* di entrambe le matrici, che valutiamo anche questa volta tramite l'utilizzo di colormap, tralasciando le configurazioni con errori altissimi ovvero 1 o 2 bit di fixed-point valutiamo visivamente l'effetto della variazione dei parametri da 3 a 7 bit per il fixed-point e da 2 a 32 per il *reduction_factor*:

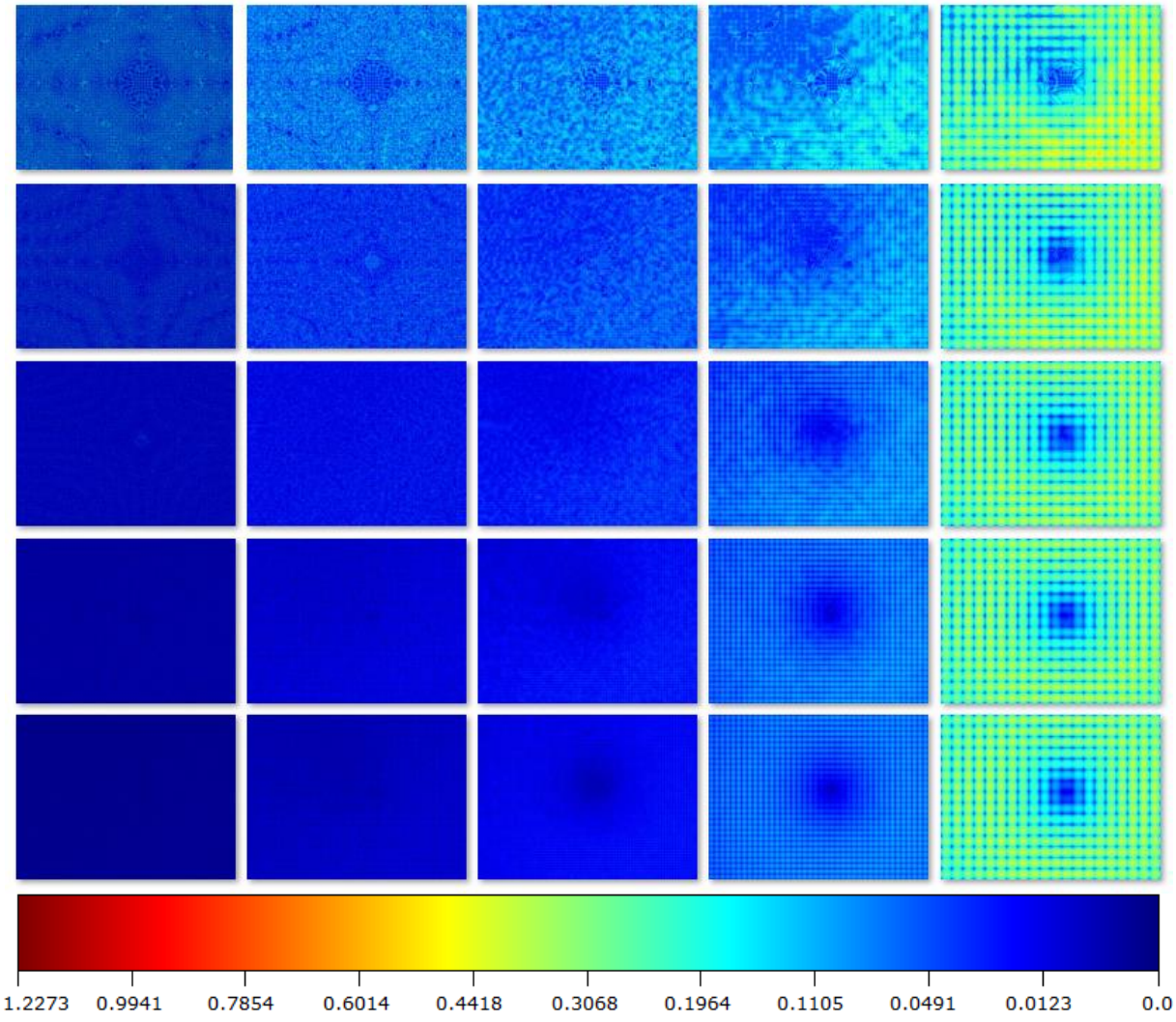


Figura 8.6.3 RMSE di entrambe le matrici x e y, reduction_factor da sinistra a destra da 2 a 32, fixed-point dall'alto al basso da 3 a 7.

Oltre tali valori infatti i risultati ottenuti sono visivamente molto simili a quelli ottenuti dalla riduzione delle matrici ma con l'uso di float.

Vediamo a questo punto le colormap delle singole matrici, mostrando solo la *left_x* dato che si ottengo risultati simili – ruotati di 90 gradi come prima – nel caso delle tabelle *y*. Notare che contrariamente a quanto potrebbe sembrare dalle immagini in miniatura, l'aumento del fixed-point non comporta un aumento dell'RMSE nella parte sinistra delle matrici, poiché l'errore medio sembra essere diminuito, ma è presente un errore in quelle zone più alto, non visibile in queste immagini.

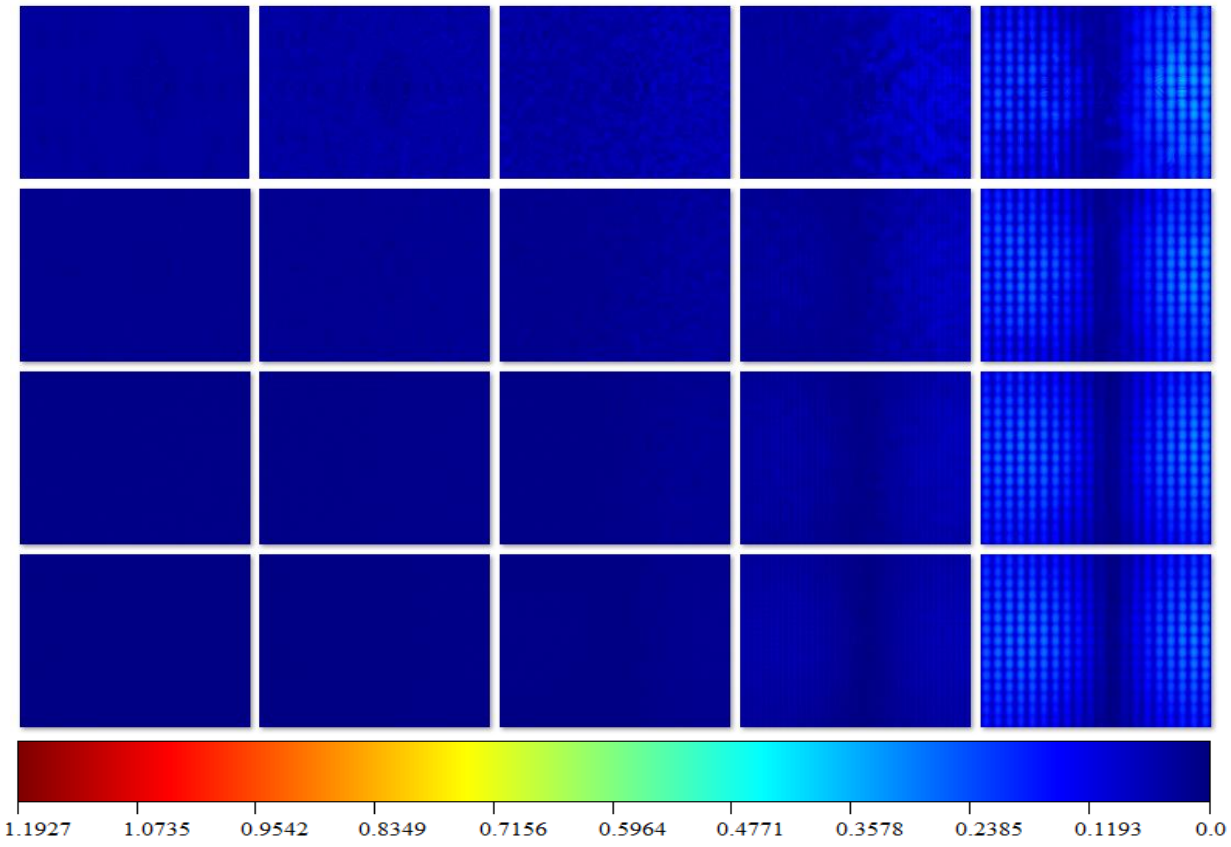


Figura 8.6.4 Colormap singole matrici, matrice left_x, reduction_factor da 2 a 32, fixed-point da 3 a 6

Ci fermiamo a 6 questa volta, per le solite e già ribadite motivazioni.

Vediamo infine le colormap riferite *all'RMSE* dei pixel dell'immagine.

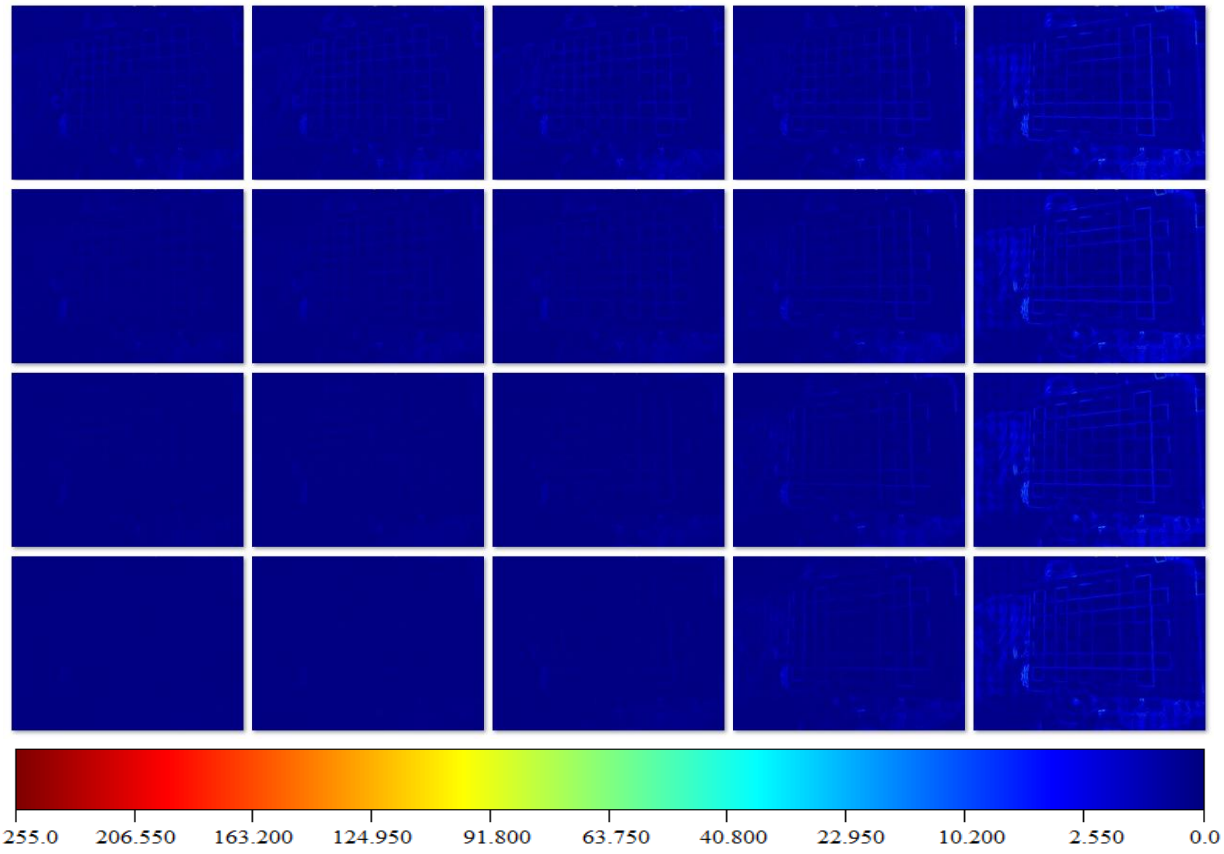


Figura 8.6.5 RMSE pixel immagine, dall'alto verso il basso fixed-point da 3 a 7, da sinistra a destra reduction_factor da 2 a 32.

Dai risultati possiamo dunque affermare che per mantenere un buon rapporto tra prestazioni e qualità della rettificazione, scegliamo di utilizzare, in una possibile reale implementazione, un *reduction_factor* di 16 e 5 bit per la parte decimale. Calcoliamo ora le dimensioni occupate dalle 4 tabelle memorizzando i displacement come relativi e, come vedremo più avanti, memorizzare in questo modo i displacement richiede 1 bit per il segno e 6 per la parte intera. I bit totali per ogni elemento sono dunque 12. Le dimensioni delle matrici ridotte con un fattore 16, nel caso di matrici di partenza 640x480, saranno di 40x30, ovvero si memorizzeranno 1200 elementi. La dimensione di una singola tabella in memoria p quindi 14400 bit. Per mantenere tutte e quattro le matrici saranno necessari 7200 byte.

8.7 Rettificazione adattiva

Con questa semplificazione abbiamo lo stesso grado di libertà della rettificazione tramite riduzione delle matrici e l'uso dei fixed-point, aggiungendo però la possibilità di variare tali parametri non per tutta l'immagine, ma diversificarli per le diverse parti dell'immagine. L'obiettivo prefissato da quest'algoritmo è quello di ottenere una qualità della rettificazione maggiore, mantenendo il quantitativo di risorse utilizzate ad un livello molto basso. Per fare ciò è stato implementato un algoritmo automatico in grado rilevare le aree che hanno più o meno necessità di essere ottimizzate e di configurare il *reduction_factor* ed il fixed-point in modo adeguato per quell'area. Resta ovviamente sempre possibile poter effettuare un *tuning* pratico per migliorare qualche zona o decidere di lascarla con più errore per risparmiare memoria. Negli esempi fatti al capitolo 7 si è utilizzato un N = 32 ed un M = 32. Questa però non rappresenta la configurazione ottimale, poiché impone di usare più sottomatrici e quindi aggiunge più righe e colonne duplicate. Nella pratica abbiamo utilizzato un fattore N = 64 e M = 32, dividendo la matrice originale in 150 sottomatrici, limitando al minimo gli sprechi. Facendo riferimento alle colormap dell'rmse sulle singole matrici, notiamo che esistono delle aree utilizzabili, e cioè ad errore basso, anche con un *reduction_factor* 32, lungo il centro ottico delle immagini. Precisamente una colonna di dimensioni 64 x 480 per le tabelle x, ed una riga di dimensioni 640 x 64 nelle tabelle y. Ciò significa che per le tabelle x delle, delle sue 150 matrici, 15 possono essere sicuramente ridotte con un fattore 32, mentre per le tabelle y, addirittura 20. A questo punto scegliamo di utilizzare per le tabelle x 5 bit per il fixed-point e per le tabelle y 6, in quanto esse presentano un errore medio più alto di quello delle y, per la nostra configurazione. Vediamo dunque le 4 configurazioni dei *reduction_factor* utilizzate per ognuna delle quattro tabelle, tenendo bene a mente che il fixed-point resta dunque uguale per tutte le

sottomatrici di una stessa tabella, per semplicità, ma che anche quest'ultimo può essere configurato in maniera adattiva come per i *reduction_factor*.

16	*16*	*16*	*16*	*16*	*32*	*16*	*16*	*16*	*16*
16	*16*	*16*	*16*	*16*	*32*	*16*	*16*	*16*	*16*
16	*16*	*16*	*16*	*16*	*32*	*16*	*16*	*8*	*8*
16	*16*	*16*	*16*	*16*	*32*	*16*	*16*	*8*	*8*
16	*8*	*8*	*16*	*16*	*32*	*16*	*16*	*8*	*8*
16	*8*	*8*	*16*	*16*	*32*	*16*	*8*	*8*	*8*
16	*8*	*8*	*16*	*16*	*32*	*16*	*8*	*8*	*8*
16	*8*	*8*	*16*	*16*	*32*	*16*	*8*	*8*	*8*
16	*8*	*8*	*16*	*16*	*32*	*16*	*8*	*8*	*8*
16	*8*	*8*	*16*	*16*	*32*	*16*	*8*	*8*	*8*
16	*8*	*8*	*16*	*16*	*32*	*16*	*8*	*8*	*8*
16	*16*	*16*	*16*	*16*	*32*	*16*	*16*	*8*	*8*
16	*16*	*16*	*16*	*16*	*32*	*16*	*16*	*8*	*8*
16	*16*	*16*	*16*	*16*	*32*	*16*	*16*	*16*	*8*
16	*16*	*16*	*16*	*16*	*32*	*16*	*16*	*16*	*16*

Configurazione dei reduction_factor delle 150 sottomatrici della left_x

16	*16*	*16*	*16*	*8*	*8*	*8*	*8*	*16*	*16*
16	*16*	*16*	*16*	*16*	*8*	*8*	*8*	*16*	*16*
16	*16*	*16*	*16*	*16*	*16*	*8*	*16*	*16*	*16*
16	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*
16	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*
16	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*
32	*32*	*32*	*32*	*32*	*32*	*32*	*32*	*32*	*32*
32	*32*	*32*	*32*	*32*	*32*	*32*	*32*	*32*	*32*
16	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*
16	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*
16	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*	*16*
16	*16*	*16*	*16*	*8*	*8*	*8*	*8*	*16*	*16*
16	*16*	*16*	*16*	*8*	*8*	*8*	*8*	*16*	*16*
16	*16*	*16*	*8*	*8*	*8*	*8*	*8*	*8*	*16*
16	*16*	*16*	*8*	*8*	*8*	*8*	*8*	*8*	*8*

Configurazione dei reduction_factor delle 150 sottomatrici della left_y

16	16	16	16	16	16	32	16	16	16
16	16	16	16	16	16	32	16	16	16
16	16	16	16	16	16	32	16	16	16
16	16	16	16	16	16	32	16	16	8
16	8	8	16	16	16	32	16	16	8
16	8	8	8	16	16	32	16	8	8
16	8	8	8	16	16	32	16	8	8
8	8	8	8	16	16	32	16	8	8
16	8	8	8	16	16	32	16	8	8
16	8	8	8	16	16	32	16	8	8
16	8	8	16	16	16	32	16	8	8
16	16	8	16	16	16	32	16	8	8
16	16	16	16	16	16	32	16	8	8
16	16	16	16	16	16	32	16	8	8
16	16	16	16	16	16	32	16	16	8

Configurazione dei reduction_factor delle 150 sottomatrici della rigth_x

16	16	16	16	8	8	8	8	16	16
16	16	16	16	16	8	8	8	16	16
16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16
32	32	32	32	32	32	32	32	32	32
32	32	32	32	32	32	32	32	32	32
16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	16	16
16	16	16	16	16	16	16	16	8	16
16	16	16	16	16	8	8	8	8	16
16	16	16	16	8	8	8	8	8	16
16	16	16	16	8	8	8	8	8	8
16	16	16	8	8	8	8	8	8	8

Configurazione dei reduction_factor delle 150 sottomatrici della rigth_y

In tabella riassumiamo ora il numero di aree per un determinato *reduction_factor* per ogni tabella e il corrispondente numero di elementi memorizzati da ogni sottomatrice che utilizza quel fattore.

reduction_factor					*elementi sottomatrice corrispondente*
8	*43*	*29*	*42*	*30*	*9x5*
16	*92*	*101*	*93*	*100*	*5x3*
32	*15*	*20*	*15*	*20*	*3x2*
	numero aree left_x	*numero aree left_y*	*numero aree right_x*	*numero aree right_y*	

Figura 8.7.1 RMSE medio singole matrici

Vediamo ora i risultati ottenuti con questa configurazione, come al solito rinunciando a mostrare l'immagine finale rettificata. Le colormap dell*'RMSE* delle singole matrici sono:

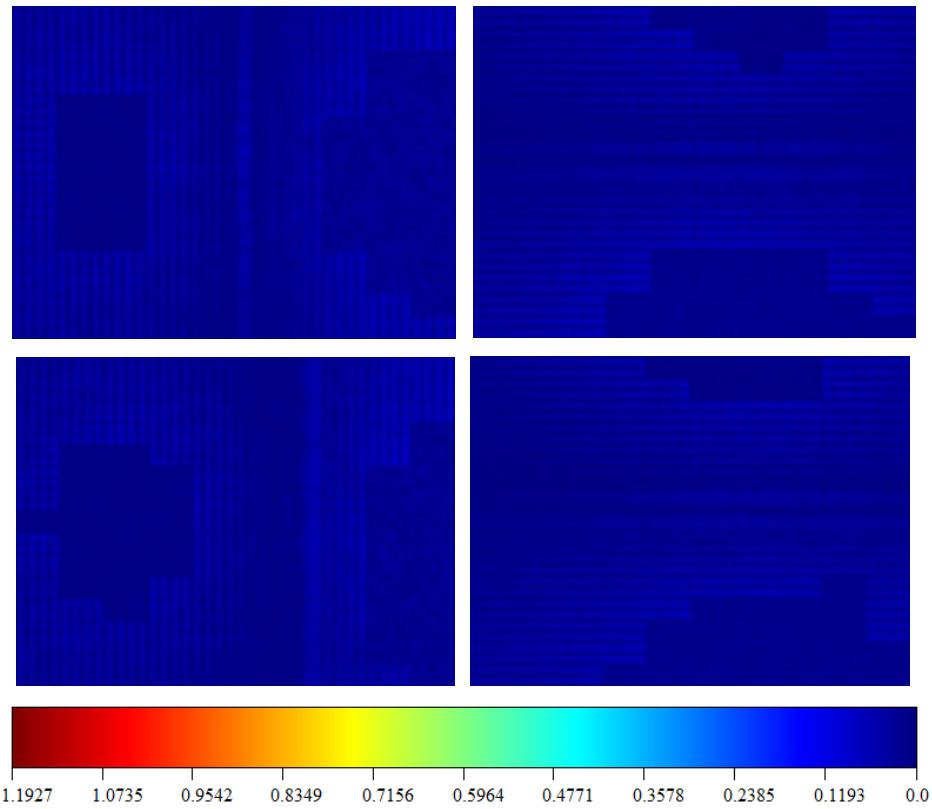


Figura 8.7.2 Colormap singole matrici, rispettivamente in alto a sinistra matrice left_x, in alto a destra matrice left_y, in basso a sinistra matrice right_x, in basso a destra matrice right_y

Quelle riferite ad entrambe le matrici invece:

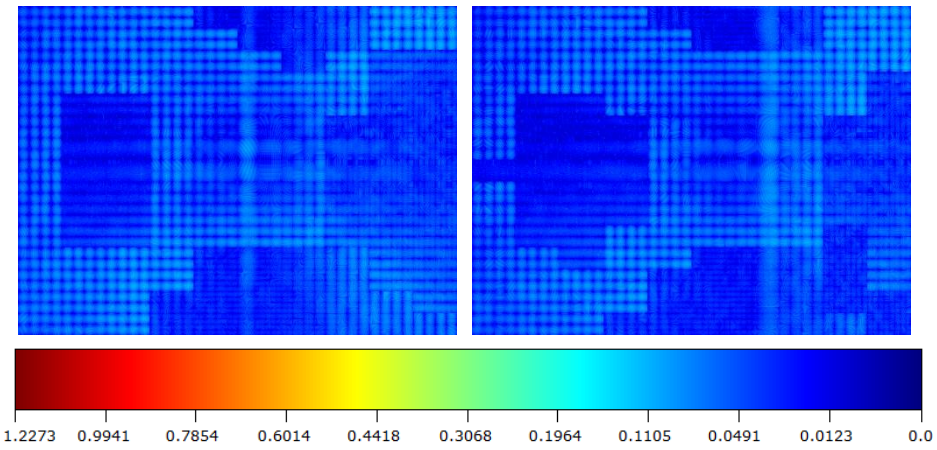


Figura 8.7.3 RMSE di entrambe le matrici x e y, a sinistra tabelle left, a destra tabelle right

A colpo d'occhio potrebbe sembrare che l'errore sia molto alto o addirittura aumentato rispetto alla configurazione indicata in precedenza come adeguata ovvero 5 bit di fixed-point e 16 come fattore di riduzione, ma così non è in quanto, guardando ai valori numerici si nota che sia l'errore medio che quello massimo sono più bassi di quelli ottenibili con quella configurazione, ed in generale, a parità di fixed-point usato nei due metodi ovvero 5 bit per le tabelle x e 6 per quelle y, questo metodo va a piazzarsi a cavallo tra 8 e 16 come errore medio e massimo delle singole tabelle che congiunte. La qualità dunque, dovrebbe essere sicuramente migliore, specialmente nei bordi, dove abbiamo abbondato nel campionamento.

RMSE				
medio	*0.037921*	*0.034479*	*0.036453*	*0.032098*
massimo	*0.114990*	*0.100525*	*0.117554*	*0.097076*
	left_x	*left_y*	*right_x*	*right_y*

Figura 8.7.4 RMSE singole tabelle

RMSE		
medio	*0.051252*	*0.048571*
massimo	*0.128020*	*0.123319*
	left_x_y	*right_x_y*

Figura 8.7.5 RMSE di entrambe le tabelle.

Anche le colormap dell'*RMSE* dei pixel migliora in modo significativo.

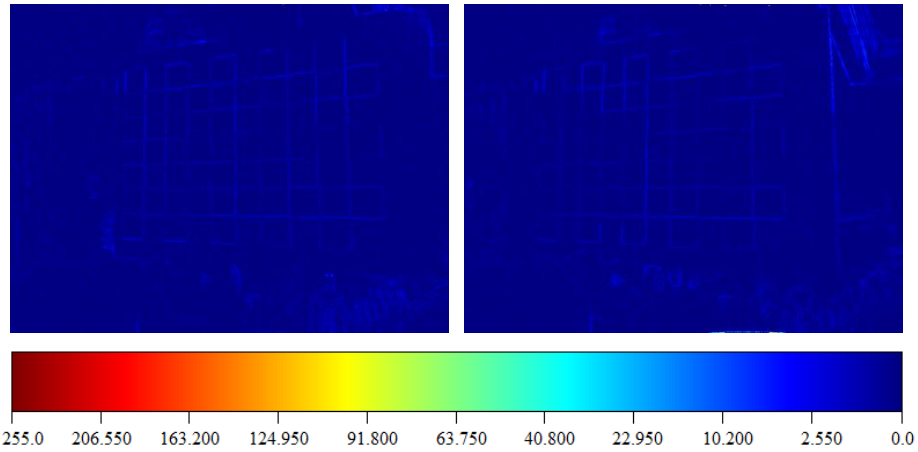


Figura 8.7.6 RMSE pixel immagini left e right.

Effettuiamo dunque ora i calcoli per vedere il quantitativo di memoria necessario a mantenere le 4 tabelle ridotte con questo metodo e la configurazione scelta. Per la tabella *left_x* come prima utilizziamo 1 bit per il segno e 6 per la parte intera del displacement relativo, a cui aggiungo i 5 bit del fixed-point abbiamo: 12 bit per ogni dato. Il numero di elementi è (15×3×2) + (92×5×3) + (43×9×5) = 3405. Dunque la dimensione occupata dalla tabella *left_x* ridotta è 3405×12 = 40860 bit. Per la tabella *left_y* utilizziamo invece 6 bit del fixed-point abbiamo: 13 bit per ogni dato. Il numero di elementi è (20×3×2) + (101×5×3) + (29×9×5) = 2940. Dunque la dimensione occupata dalla tabella *left_x* ridotta è 2940×13 = 38220 bit. Per la tabella *right_x* utilizziamo invece

nuovamente 5 bit del fixed-point abbiamo: 12 bit per ogni dato. Il numero di elementi è (15×3×2) + (93×5×3) + (42×9×5) = 3375. Dunque la dimensione occupata dalla tabella *left_x* ridotta è 3375×12 = 40500 bit. Per la tabella *right_y* utilizziamo invece 6 bit del fixed-point abbiamo: 13 bit per ogni dato. Il numero di elementi è (20×3×2) + (100×5×3) + (30×9×5) = 2970. Dunque la dimensione occupata dalla tabella *left_x* ridotta è 2970×13 = 38610 bit. In totale occupiamo quindi 158190 bit, ovvero 19,774 kbyte. A cui dobbiamo aggiungere infine, i 150 *reduction_factor* e i 150 *nbit* dei fixed-point per ogni tabella. Dunque, per i *reduction_factor* utilizziamo 5 bit se memorizzati per intero, 3 se memorizzati come esponente della potenza di 2, e per gli nbit usiamo 3 bit. Quindi in totale, nel caso peggiore, aggiungiamo 375 byte per i *reduction_factor* e 225 byte per gli *nbit*. Un quantitativo come già predetto sicuramente maggiore, ma qualitativamente superiore in ogni punto del metodo precedente, e comunque compatibile con sistemi hardware con risorse limitate.

8.8 Dimensioni buffer immagini acquisite

Fino ad ora si è sempre considerata l'immagine distorta come se fosse memorizzata per intero e rettificata tutta d'un colpo, ma non è così. In hardware infatti, la poca memoria disponibile deve essere ben gestita, ed infatti l'immagine viene memorizzata su di un buffer circolare una riga alla volta. In questo scenario dobbiamo calcolare la dimensione del buffer, e per fare ciò vi è la necessità di conoscere quale sia il displacement relativo massimo e quello minimo, tra tutte e quattro le tabelle, dato che dal punto che stiamo per rettificare nell'immagine non distorta dobbiamo poter andare a prendere i relativi 4 pixel adiacenti al punto dell'immagine distorta indicato nella tabella dei displacement, che può appunto trovarsi al massimo *y_max* coordinate *y* verso l'alto e *y_min* coordinate *y* verso il basso. Vediamo con un'immagine:

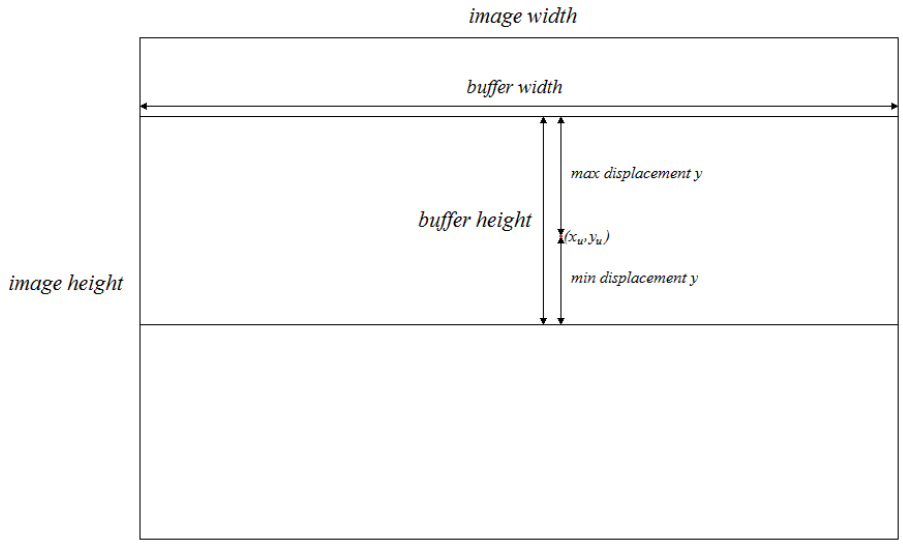


Figura 8.8.1 Buffer pixel immagine distorta da rettificare. (x_u, y_u) coordinata del punto che stiamo per rettificare nell'immagine non distorta.

Con il sistema utilizzato si è visto che il che il displacement relativo minimo è, mentre il massimo è 61.870220. Sapendo che abbiamo bisogno di 8 bit per l'intensità di ogni pixel, e che la nostra *image width* è 640, e che la *buffer height* è (61.870220 + | -53.317230 |) = 115,18745 e quindi 116, abbiamo che il buffer deve essere grande almeno 640 × 116 × 8 bit = 593920 bit ovvero 74240 byte.

Capitolo 9

Conclusioni

9.1 Conclusioni

In questo lavoro è stato mostrato come poter eseguire la rettificazione mediante semplificazioni del metodo originale nei dispositivi embedded sprovvisti della possibilità di usare i float e sprovvisti di grandi quantitativi di memoria interna. Abbiamo affrontato e risolto problematiche di natura varia tramite l'utilizzo di soluzioni già proposte da altri autori e infine proposto nuove soluzioni a determinati problemi, come ad esempio gli effetti di bordo. Abbiamo visto, studiato e analizzato gli effetti delle semplificazioni e di ogni singolo parametro. Siamo poi stati in grado di elaborare un nuovo metodo che, tramite maggiori accorgimenti e una giusta configurazione, può rappresentare una valida alternativa per aumentare la qualità generale della rettificazione senza perdere di vista l'obbiettivo principale, ovvero ridurre la memoria utilizzata. Inoltre questo nuovo metodo potrebbe essere migliorato, eliminando o riducendo la necessità di replicazione, che è il vero limite per la riduzione del quantitativo di memoria richiesto. In alternativa, si potrebbe pensare di bufferizzare anche l'uso delle tabelle, che col nuovo metodo diventano l'unica informazione da avere per ricavare il *displacement* per un'area di NxM pixel. Ciò permetterebbe infatti una riduzione delle dimensioni occupate dalle tabelle in memoria davvero considerevole, e con una rettificazione

adattiva magari, si potrebbe ridurrebbe l’errore ad un valore molto basso, quasi prossimo allo zero. Concludiamo quindi affermando che gli obiettivi prefissati sono stato raggiunti.

Ringraziamenti

Vorrei ringraziare infine, il Prof. Stefano Mattoccia, per la grande disponibilità e per avermi dato la possibilità di lavorare in un ambito molto interessante, e mai incontrato durante il periodo di studi universitario, che mi ha fatto acquisire nuove conoscenze e rinforzato quelle precedenti. Ringrazio infine i miei genitori per avermi dato la possibilità di proseguire gli studi nonostante i sacrifici, e di avermi sempre sostenuto al meglio.

Bibliografia

[1] A. Kaehler G. Bradski, Learning OpenCV, O'Reilly, 2008

[2] OpenCv Reference Manual 3.1, 21 Dec. 2015

[3] OpenCv Source Code, 3.1, sources\ samples\ cpp\ tutorial_code\ calib3d\ camera_calibration\ camera_calibration.cpp, from http://docs.opencv.org/3.1.0/d4/d94/tutorial_camera_calibration.html

[4] OpenCv Source Code, 3.1, sources\ samples\ cpp\ tutorial_code\ stereo_calib.cpp

[5] Tesi D. Nanni, "Analisi di algoritmi di rettificazione stereo per fpga", AA 2009/10

[6] S. Mattoccia, "Stereo vision algortihms and applications", www.vision.deis.unibo.it/smatt