Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Analisi del mapping su architettura SIMPil di un sistema di riconoscimento automatico dei segnali stradali

M. Liotta, A. Gentile
S.Vitabile, F. Sorbello

RT-ICAR-PA-02-01 **dicembre 2002**

Consiglio Nazionale delle Ricerche, Istituto di Calcolo e Reti ad Alte Prestazioni (ICAR) – Sede di Cosenza, Via P. Bucci 41C, 87036 Rende, Italy, URL: *www.icar.cnr.it*
– Sezione di Napoli, Via P. Castellino 111, 80131 Napoli, URL: *www.na.icar.cnr.it*
– Sezione di Palermo, Viale delle Scienze, 90128 Palermo, URL: *www.pa.icar.cnr.it*

Consiglio Nazionale delle Ricerche
Istituto di Calcolo e Reti ad Alte Prestazioni

Analisi del mapping su architettura SIMPil di un sistema di riconoscimento automatico dei segnali stradali

M. Liotta[2], A. Gentile[2]
S.Vitabile[1], F. Sorbello[2]

Rapporto Tecnico N.: **RT-ICAR-PA-02-01**

Data: **dicembre 2002**

---

[1] Istituto di Calcolo e Reti ad Alte Prestazioni, ICAR-CNR, Sezione di Palermo
[2] Università degli Studi di Palermo Dipartimento di Ingegneria Informatica

Analisi del mapping su architettura SIMPil

di un sistema di riconoscimento automatico

dei segnali stradali

M. Liotta– A. Gentile
S.Vitabile – F. Sorbello

Rapporto tecnico CE.R.E 1/02

INDICE

Sommario

Nel presente lavoro si effettua l'analisi di un sistema, da implementare sull'architettura parallela SIMPil (SIMD Pixel Processor), che permetta l'estrazione ed il riconoscimento di segnali stradali da scene esterne.

Si utilizza la segmentazione di oggetti presenti in immagini digitali a colori di scene naturali, scegliendo all'interno del processo di aggregazione dei pixel, rappresentati nello spazio HSV, una soglia dinamica che permetta di ridurre l'instabilità della tinta, nelle scene reali, dovuta a variazioni della luminosità esterna.

1. Introduzione

L'obiettivo principale di un sistema di riconoscimento automatico di segnali stradali (Automatic Road Signs Recognition Systems, { $A(RS)^2$}) è quello di individuare uno o più segnali stradali a partire da immagini digitali complesse provenienti da una videocamera CCD, montata su un veicolo in moto lungo le strade, in modo tale che possa essere utilizzato a scopi di assistenza alla guida, fornendo, per esempio, dei messaggi di pericolo, oppure per la navigazione autonoma di veicoli mobili.

Nella progettazione e nello sviluppo di sistemi di tal tipo, un ruolo determinante riveste una buona segmentazione, mediante la quale, un'immagine viene divisa nelle sue parti più significative.

Tale processo, può essere reso più o meno robusto per mezzo di uno studio attento delle problematiche legate ai difetti nell'acquisizione dell'immagine ed in generale a tutte le possibili cause naturali o artificiali che influenzano la corretta individuazione delle regioni di interesse. Questo compito è molto difficile se si considerano sia la complessità delle scene esterne, sia la variazione delle condizioni di luminosità, il controllo delle quali è molto complesso, considerato che l'intensità della luce dipende dall'ora del giorno, dalla stagione e dalle condizioni ambientali; a ciò si aggiunge che la posizione del segnale non è sempre quella frontale e in più che nei segnali possono essere presenti ombre dovute ai segnali circostanti.

Per massimizzare la robustezza del sistema si utilizzano delle metodologie che consentono di affinare il processo di riconoscimento delle forme attraverso le caratteristiche del colore, tenendo anche conto dei problemi legati all'influenza dei riflessi di luce sugli oggetti e delle mutevoli condizioni dell'ambiente esterno, che comportano variazioni non prevedibili a priori delle proprietà locali a globali delle immagini.

I sistemi utilizzati per l'acquisizione delle immagini digitali in genere utilizzano un sistema di rappresentazione del colore fondato sull'utilizzo dei tre colori primari rosso, verde e blu (rappresentazione cromatica RGB). Questa rappresentazione, però, non è la più adatta per la segmentazione di scene naturali, poiché una variazione dell'intensità luminosa dell'ambiente si riflette in maniera lineare su tutte e tre le componenti RGB. A tal fine si utilizza la rappresentazione HSV (Hue, Saturation, Value), perché consente di avere una sola componente dipendente dalla luminosità della scena, mentre le altre due componenti sono legate a caratteristiche intrinseche del colore.
Se da un lato, però, l'uso della rappresentazione HSV permette di mettere a punto sistemi di segmentazione più robusti, dall'altro la robustezza della rappresentazione HSV, rispetto alle variazioni dell'intensità luminosa, vale soltanto in determinate condizioni di acquisizione; le proprietà del colore degli oggetti risultano, infatti, invarianti rispetto alla variazione della luminosità esterna, soltanto sotto due precise condizioni: la sorgente di luce deve essere bianca e il colore degli oggetti della scena deve avere una saturazione abbastanza elevata, negli altri casi si deve tenere conto di un cambiamento non prevedibile della tinta del colore degli oggetti, che rende necessaria l'applicazione di metodologie per la corretta segmentazione della scena, per questo motivo l'algoritmo di segmentazione implementato tiene conto della saturazione dell'immagine per determinare la forma del sottospazio di interesse nello spazio HSV dei colori.
Una volta estratta la regione di interesse è possibile effettuare una classificazione mediante lo studio sulla loro forma e sul loro contenuto, per mezzo delle numerose procedure che studiano la morfologia delle regioni estratte dalla scena, che utilizzano dei codici che descrivono il contorno, il calcolo di eventuali simmetrie rispetto ad assi, reti neurali ed altri ancora.

Un altro aspetto da evidenziare è che una caratteristica chiave della elaborazione delle immagini è che una notevole parte del tempo di elaborazione viene impiegata ripetendo uno stesso insieme di operazioni su un flusso continuo di dati; poiché l'interazione tra i dati contigui è poca o nulla, vi è un elevato *parallelismo sui dati*; per sfruttare questa peculiarità, si utilizzano i PMMS (Portable MultiMedia Supercomputing), i quali utilizzano l'intrinseco parallelismo di dati delle applicazioni multimediali utilizzando molti processori semplici invece che pochi processori potenti: fra i modelli sviluppabili per i PMMS, le architetture *SIMD* (Single Input Multiple Data) risultano le migliori

grazie all'enorme grado di parallelismo che queste permettono di implementare, come si evidenzia nella Fig 1.1.

Esiste già una implementazione del sistema {A(RS)^2} su processore Intel III 550 MHz sotto Linux [1]; lo scopo di questo lavoro è quello di analizzare le caratteristiche di un sistema { A(RS)^2} da implementare, successivamente, sull'architettura SIMPil (SIMD Pixel Processor) [2], sviluppato dal PICA Research Group del Georgia Institute of Technology.

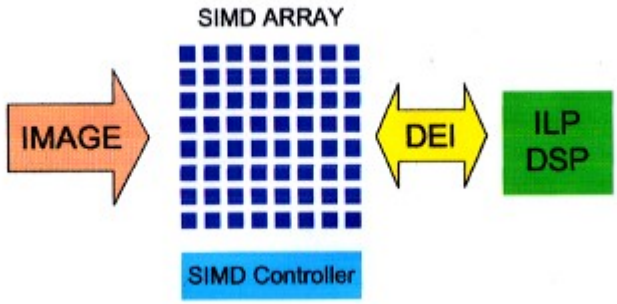


Fig 1. 1 [3]: Architettura PMMS

Nella sezione successiva vengono descritte le caratteristiche fondamentali e la struttura dell'architettura SIMPil; nella sezione 3 si analizza la struttura generale ed il flusso logico dell'implementazione del sistema {A(RS)^2}; nella sezione 4 si analizzano in dettaglio le funzioni necessarie per parallelizzare il sistema {A(RS)^2} già esistente, al fine di rendere possibile un'implementazione su SIMPil; nella sezione 5 è presente l'implementazione della parte del sistema riguardante l'algoritmo di conversione delle componenti del colore dallo spazio RGB allo spazio HSV, in Appendice A è presente il relativo codice Assembler e in Appendice B un approfondimento sull'algoritmo di divisione utilizzato.

2. Architettura SIMPil

Il candidato principale per i PMMS, tra le architetture SIMD, è SIMPil (SIMD pixel processor), un sistema sviluppato dal PICA Research Group della Georgia Tech che permette di raggiungere i requisiti necessari al real-time nelle applicazioni multimediali grazie ad un elevato equilibrio tra

elaborazione, immagazzinamento e comunicazione superiore a quello delle architetture DSP; utilizza, infatti, i benefici del parallelismo dovuto all'array SIMD, collegamenti corti ed una microarchitettura finalizzata a fornire un notevole aumento dell'efficienza energetica [3],[4],[5],[6].

La struttura di SIMPil comprende un array SIMD di processori digitali (PE) che ottimizzano l'elaborazione di un flusso di dati e un array di dispositivi optoelettronici viene integrato in cima al PE e con questo viene interfacciato elettricamente alla circuiteria analogica.
L'architettura SIMPil prevede un'area di I/O integrata che risulta essere la principale sorgente per l'I/O dei dati, utilizza un supporto per il *processing in place* consentendo un throughput elevato ed un piccolo utilizzo di memoria grazie, appunto, ad un'efficiente interfaccia tra sensori e processori; questa scelta è molto vantaggiosa perché si evitano le fasi di inattività dovuti all'attesa del fetch dalla memoria, visto che un unico flusso di dati è distribuito tra i vari PE, rendendoli effettivamente operativi infatti, l'elaborazione sui pixel che compongono l'immagine viene eseguita quando arriva il flusso di dati stesso e, quindi, non è necessario il trasporto dei dati a partire da un buffer centralizzato: in tal modo è possibile raggiungere prestazioni molto elevate, dell'ordine di 500-1500 Gops/sec.
L'area di I/O permette un processamento vettoriale 2D: l'immagine bidimensionale è correlata spazialmente con l'area di I/O.

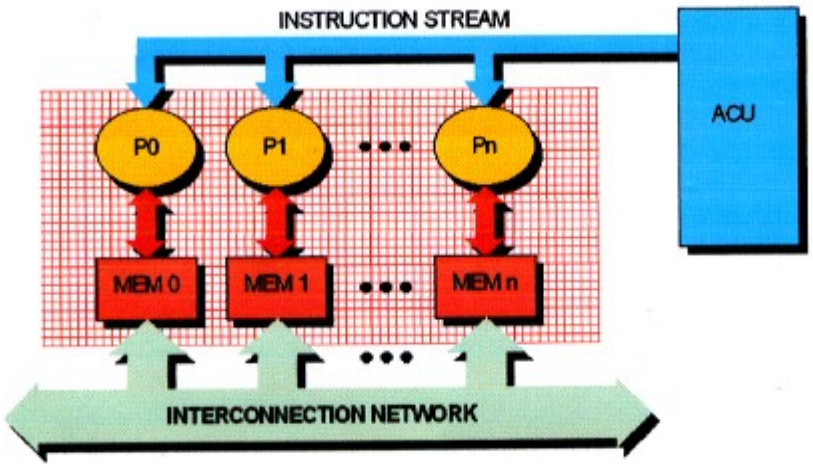


Fig 2.1 [4]: Organizzazione di un'architettura parallela di tipo SIMD

Il programma è immagazzinato nella ACU (Array Control Unit) e tramite broadcast sui nodi, ogni processore del sistema riceve un'istruzione (single istruction stream) e la esegue sui dati localmente (multiple data stream); ogni nodo, utilizzando la rete di interconnessione, può scambiare dati con gli altri 4 nodi con cui è interconnesso con i suoi 4 vicini per mezzo di una struttura a mesh, può anche

assumere una configurazione toroidale, così da utilizzare uno schema di comunicazione più potente rispetto alle tradizionali reti NEWS (North-East-West-South).

L'architettura interna del singolo PE è del tripodi Fig 2.2 che mostra come l'architettura SIMPil sia costituita da un array di processori SIMD in cui ogni PE é di tipo RISC, ha unità specifiche per operazioni di tipo SIMD e meccanismi particolari per l'area di I/O del flusso dati.
Le principali unità funzionali sono:

- ALU e SHIFTER.
- MACC (Multiply-acumulator) contenente un accumulatore a doppia precisione.
- Registers File: 16 registri a 16 bit (16x16).
- Memoria locale contenente un numero di locazioni a 16 bit pari al parametro LOCAL_MEM_SIZE (LOCAL_MEM_SIZE x16).
- Pixel Register: registri nei quali viene memorizzata l'immagine. Il numero dei registri dipende dal numero di pixel contenuto nel singolo preocessore (PPE); in totale ci sono PPE registri a 8 bit (PPEx8).
- Unità di comunicazione e di I/O seriale.

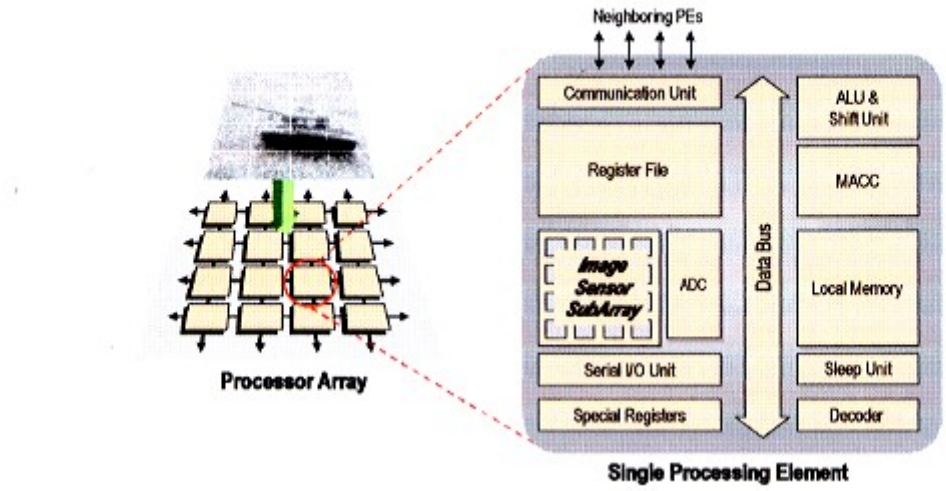


Fig 2.2 [3]: Diagramma dell'array processor e del PE della struttura SIMPil

Tramite l'unità di comunicazione, ogni PE può comunicare con i 4 processori adiacenti (North, East, West, South) e, sebbene la maggior parte delle architetture SIMD utilizzino registri speciali associati a ciascuno dei 4 nodi vicini (i NEWS registers), l'architettura SIMPil prevede che il contenuto dei registri sia sorgente e destinazione per i nodo adiacenti.

L’istruzione SAMPLE campiona simultaneamente tutti i valori sui rivelatori e li rende utilizzabili per elaborazioni successive.
Il modello di esecuzione SIMD permette di campionare in un singolo ciclo l’immagine in ingresso proiettata sui vari nodi.

La struttura interna della ACU (Array Control Unit) contiene:

- ALU
- Moltiplicatore
- 8 File register a 16 bit (8x16)
- Memoria con 64 locazioni a 16 bit (64x16)

3. Analisi generale del sistema {A(RS)^2}

3.1 Introduzione

Come già detto, esiste già un’implementazione, in linguaggio C, del sistema {A(RS)2} [7] su un processore Intel III, la quale presenta, come caratteristica chiave, un elevato parallelismo sui dati dovuto alla ripetizione di uno stesso insieme di istruzioni riguardanti i pixel dell’immagine, i quali, peraltro, hanno una bassa interazione; questa peculiarità permette di scomporre l’immagine di partenza in sotto-immagini, ognuna interna ad un solo PE di SIMPil e di eseguire tutte le istruzioni parallelamente tra i vari PE; l’ipotesi base che si farà nel seguito è quella di dividere l’immagine di partenza in sotto-immagini e far elaborare la singola porzione di immagine ad un processore dell’array, secondo il seguente schema:

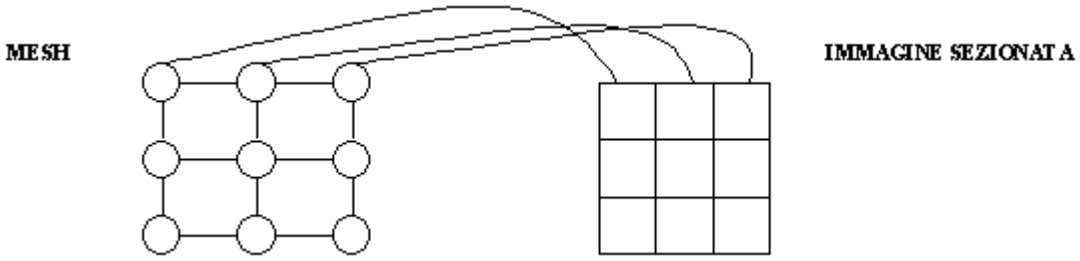


Fig 3. 1: Suddivisione dell'immagine tra i PE

Nel presente capitolo, si analizza la struttura dell'implementazione seriale, già esistente, del sistema {A(RS)2} e si lascia alla sezione 4 l'analisi delle funzioni necessarie per la parallelizzazione del sistema.

3.2 Descrizione del sistema

Lo scopo del sistema, come già detto, è quello di riconoscere ed estrarre da scene naturali i segnali stradali; affinché ciò sia possibile, una volta effettuata l'acquisizione dell'immagine, si deve effettuare una prima fase di pre-elaborazione per ridurre il più possibile il rumore puntuale legato all'acquisizione stessa; la seconda fase è quella di segmentazione nella quale si aggregano i pixel all'interno della regione in base ad un criterio di similitudine; nella terza fase, si effettua un'analisi sulla forma dell'area precedentemente segmentata.

Il diagramma di flusso del sistema generale è quello di Fig 3.2.

Acquisita l'immagine, quindi, le fasi da effettuare sono:

1. *Pre-elaborazione*:
    a. riduzione del rumere puntuale
    b. conversione RGB $\rightarrow$ HSV
2. *Segmentazione*:
    a. sogliatura dello spazio HSV
    b. filtro per punti isolati
    c. scelta del pixel seme

d. aggregazione dei pixel

3. *Riconoscimento*:

   a. estrazione delle regioni di interesse

   b. pre-classificazione

Nel seguito si analizzano le diverse fasi e le relative sotto-fasi.

3.2.1 Pre-elaborazione

Questa fase ha lo scopo di migliorare la qualità dell'immagine acquisita riducendo i disturbi dovuti alle non perfette condizioni di acquisizione.

In riferimento alla Fig 3.2, il blocco relativo alla pre-classificazione può essere suddiviso nei sotto-blocchi rappresentati in Fig 3.3, in cui le operazioni presenti nei due sotto-blocchi sono applicati uniformemente all'intera immagine che si suppone essere composta da 320x240 pixel.

Il primo sotto-blocco ha lo scopo di eliminare il rumore puntuale presente su pochi pixel, per mezzo di un apposito filtro non lineare (SNN, Symmatrical Nearest Neighbour), il quale fa in modo che le regioni che hanno delle componenti RGB uniformi siano più omogenee possibile: un tale procedimento prende il nome di *smoothing*.

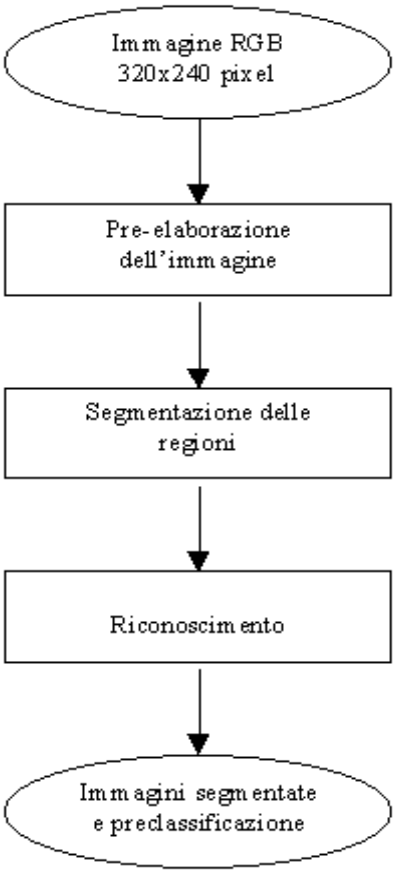


Fig 3. 2: Diagramma di flusso del sistema

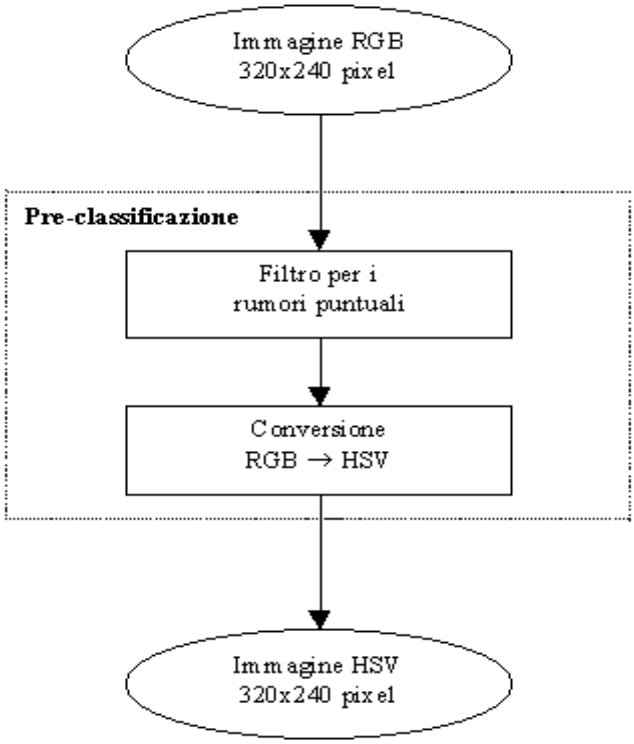


Fig 3. 3: Sotto-blocchi del blocco *pre-classificazione*

Si faccia riferimento ad una struttura 3x3, del tipo di Fig. 3.4 centrata sul pixel c_0.

C_1	C_2	C_3
C_4	C_0	C_5
C_6	C_7	C_8

Fig 3. 4: Struttura di 3x3 pixel centrata su C_0

Il filtro SNN permette di ottenere un effetto di *smoothing* dell'immagine e nello stesso tempo non pregiudicare i contorni, utilizzando il seguente meccanismo: indicando con c_i la posizione di ogni vicino,$f(c_i)=(r_i, g_i, b_i)$ denota i valori di R,G,B nella posizione c_i, dato il pixel centrale c_0 si considerano le coppie $(c_1, c_8),(c_2, c_7),(c_3, c_6),(c_5, c_4)$. Il pixel centrale viene sostituito dalla media pesata delle 4 coppie determinate.

Posto $v_i=(r_i, g_i, b_i)$ con $1<i<4$, i vettori v_i vengono calcolati secondo lo schema descritto di seguito:

$$v_1 = f(c_1) \text{ se } \|f(c_0) - f(c_1)\| \leq \|f(c_0) - f(c_8)\|$$
$$= f(c_8) \text{ altrimenti}$$

$$v_2 = f(c_2) \text{ se } \|f(c_0) - f(c_2)\| \leq \|f(c_0) - f(c_7)\|$$
$$= f(c_7) \text{ altrimenti}$$

$$v_3 = f(c_3) \text{ se } \|f(c_0) - f(c_3)\| \leq \|f(c_0) - f(c_6)\|$$
$$= f(c_6) \text{ altrimenti}$$

$$v_4 = f(c_4) \text{ se } \|f(c_0) - f(c_4)\| \leq \|f(c_0) - f(c_5)\|$$
$$= f(c_5) \text{ altrimenti}$$

dove $\|f(c_i) - f(c_j)\|$ in accordo con la metrica usata nello spazio RGB vale :

$$\|f(c_i) - f(c_j)\| = \sqrt{(r_i - r_j)^2 + (g_i - g_j)^2 + (b_i - b_j)^2}$$

calcolati i valori v_i la tripla RGB del pixel centrale c_0 viene sostituita da :

$$f(c_0) = \frac{1}{4}\left[\sum_{i=1}^{4} r_i, \sum_{i=1}^{4} g_i, \sum_{i=1}^{4} b_i\right]$$

Per maggiori approfondimenti si veda, nella sezione 4, il paragrafo riguardante la funzione "**Filtro_Snn**".

Il secondo sotto-blocco esegue la conversione delle componenti del colore dallo spazio RGB allo spazio HSV [8], questo si rende necessario, perché, se da un lato, la maggior parte dei sistemi utilizzati per l'acquisizione e la visualizzazione delle immagini digitali forniscono una rappresentazione delle immagini nello spazio dei colori RGB, cioè secondo le componenti dei tre colori primari il rosso (R), il verde (G), il blu (B), dato che ogni colore può essere visto come una combinazione dei tre colori primari, dall'altro questo sistema è poco efficace nell'elaborazione; di conseguienza il sistema RGB risulti poco adatto al caso della segmentazione di scene naturali, visto che questa risulta fortemente influenzata dalle condizioni di luminosità; ciò accade perché le intensità delle tre immagini vengono alterate in funzione della luminosità, indipendentemente le una dalle altre e quindi si altera la proporzione tra le tre componenti dei tre colori primari.

Nello spazio dei colori HSV si caratterizza l'immagine per mezzo della intensità luminosa, utilizzando la componente V (Value), e per mezzo della crominanza [9] costituita da due elementi: la tinta e la saturazione rappresentate rispettivamente dalle componenti H (Hue) e S (Saturation).

La tinta rappresenta il colore dominante dell'oggetto, la saturazione è relativa alla purezza del colore, cioè la quantità di luce bianca mescolata con la tinta: il grado di saturazione è tanto più elevato quanto minore è la quantità di bianco.

A differenza del sistema RGB in cui al variare della luminosità varia il valore della crominanza, qui si può modificare liberamente l'intensità dato che questa è separata dalle componenti di crominanza: è, quindi, possibile realizzare dei sistemi più robusti rispetto alla luminosità esterna.

Quanto detto risulta vero soltanto se:

1. la sorgente di luce è bianca

2. il colore degli oggetti presenti nella scena ha un valore di saturazione piuttosto elevato

se queste due condizioni non sono verificate, è possibile che la tinta del colore degli oggetti presenti nella scena, cioè il valore H, possa variare in maniera non prevedibile.

Per maggiori approfondimenti si veda il paragrafo, nella sezione 4, riguardante la funzione "**ConversioneRGB_HSV**".

3.2.2 Segmentazione

L’obiettivo di questa fase è quello di partizionare la scena in regioni al fine di potere estrarre oggetti o altre entità per le successive fasi della elaborazione [10].

Il blocco relativo alla segmentazione può essere scomposto nei sotto-blocchi riportati in Fig 3.5 [1]. Il primo sotto-blocco, a sinistra e a destra, effettua un filtro brutale, per il rosso e per il blu rispettivamente, basato sulle coordinate HSV dei singoli pixel al fine di produrre una maschera che esprima il soddisfacimento o meno del filtro. Vengono fissati dei valori H, S, V e i relativi intervalli, aventi questi come estremi, all’interno dei quali devono appartenere i valori di H, S e V di ogni pixel affinché il pixel stesso possa essere classificato come rosso e blu rispettivamente. In questo modo si crea una maschera binaria in cui il valore è 1 se il corrispondente pixel, a seconda del caso, è da considerarsi rosso o blu, è 0 altrimenti.

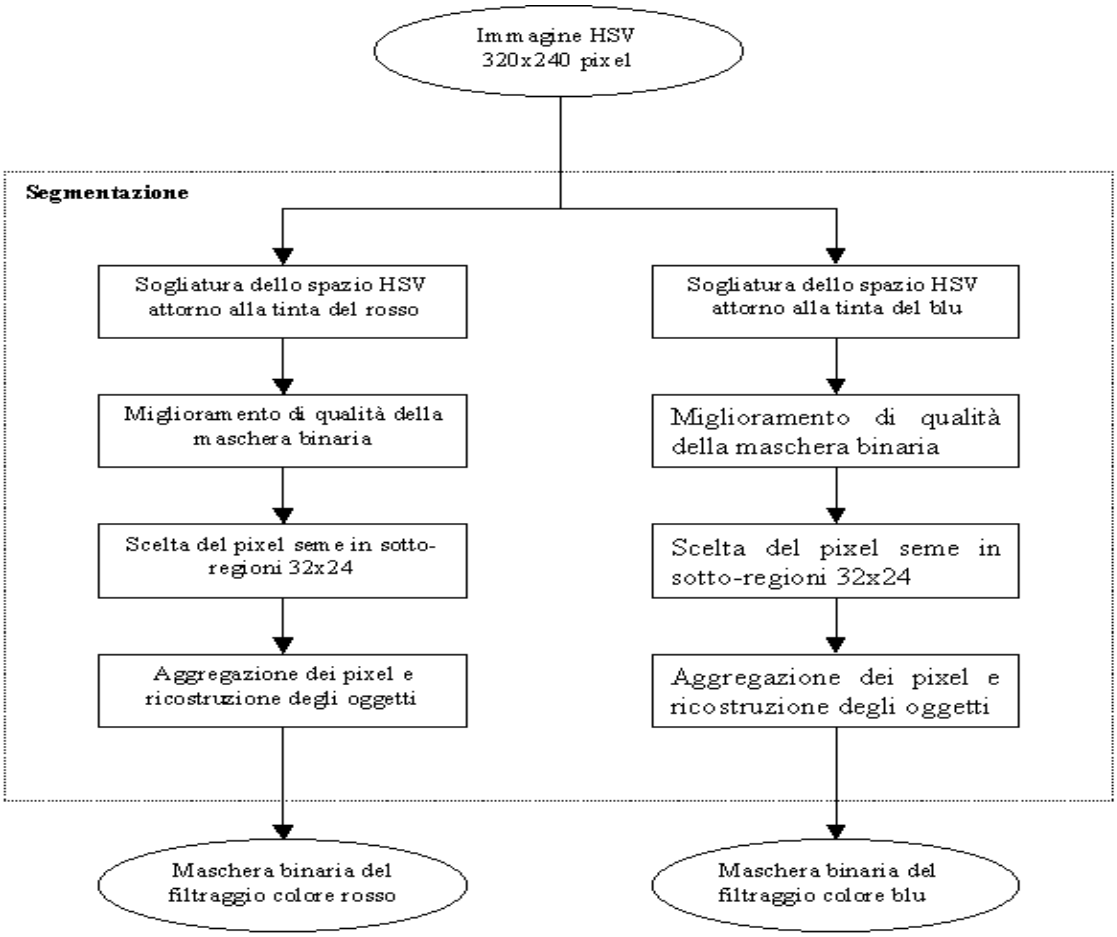


Fig 3. 5: Sotto-blocchi del blocco *segmentazione*

Per maggiori approfondimenti si veda il paragrafo, nella sezione 4, riguardante la funzione "**Filtro_nello_spazio_HSV_rosso**" e "**Filtro_nello_spazio_HSV_blu**".

Il secondo sotto-blocco ha lo scopo di migliorare la qualità della maschera binaria per aumentare l'efficienza degli algoritmi, in particolare, sono stati implementati due filtri per eliminare le zone di rumore sia nello sfondo dell'immagine (zone bianche della maschera), sia sulle forme trovate (zone nere della maschera). Per maggiori approfondimenti si veda il paragrafo, nella sezione 4, riguardante la funzione "**Filtro_Neri_isolati**" e "**Filtro_Bianchi_isolati**".

Il terzo sotto-blocco ha lo scopo di individuare il pixel seme, necessario per la tecnica del *region growing* [11] descritta dopo, da considerare per la fase di accrescimento all'interno di ogni sotto-

regione corrispondente alla sotto-immagine presente nel singolo processore: il seme viene individuato come media dei valori RGB dei pixel contenuti nelle sotto-regioni.
Per maggiori approfondimenti si veda, nella sezione 4, il paragrafo riguardante la funzione "**Seme_Rosso**".
Il quarto sotto-blocco, infine, implementa il "*region growing*", cioè quella tecnica che consiste nel raggruppare i pixel o le sotto-regioni in regioni più ampie al fine di estrarre delle immagini dall'intera regione: a partire dal pixel seme si aggregano, sempre all'interno delle sotto-immagini, i pixel vicini aventi proprietà simili, secondo il criterio di similitudine basato sulla distanza euclidea nello spazio HSV: il pixel seme ed il pixel vicino risultano simili se la distanza euclidea nello spazio HSV è minore di una determinata soglia; la soglia, con cui si giudica la similitudine tra i due pixel, si fa dipendere in maniera non lineare dal valore di saturazione del pixel seme, secondo la relazione:

$$a = k - sen\ (S)$$

essendo a è il valore della soglia, k un parametro fissato sperimentalmente e S il valore di saturazione del pixel seme, in modo tale che, per bassi valori di saturazione, il criterio di similitudine diviene più restrittivo, rendendo l'algoritmo più robusto rispetto alla instabilità della tinta proprio intorno ai bassi valori di saturazione: dati due pixel seme, caratterizzati da diversi valori di saturazione, è possibile individuare delle regioni diverse nello spazio HSV a pari valore di Vin cui al pixel seme con minore saturazione vengono aggregati i pixel vicini spazialmente che hanno distanza nello spazio HSV minore, rispetto al caso del pixel seme con saturazione più elevata; nel primo caso, quindi, vengono aggregati i pixel che hanno tinta molto simile, nel secondo caso, invece, vengono aggregati anche pixel che hanno tinta piuttosto diversa.
Per maggiori approfondimenti si veda, nella sezione 4, il paragrafo riguardante la funzione "**Growing**".

3.2.3 Riconoscimento

Questa fase ha lo scopo di estrarre la regione di interesse dall'intera immagine e di determinare il tipo di segnale stradale.

Il blocco relativo alla segmentazione può essere scomposto nei sotto-blocchi rappresentati in Fig 3.6.

Il primo sotto-blocco, a sinistra e a destra, ha come scopo la determinazione delle coordinate massime e minime delle regioni segmentate; a tal fine si visitano, ricorsivamente, tutti i pixel e si aggiornano delle apposite variabili che, alla fine, conterranno le coordinate massime e minime del rettangolo che circoscrive la regione.

L'algoritmo agisce sulla maschera ottenuta dal filtraggio sul colore:

1. Individuato un pixel nero, quindi facente parte di una regione, si memorizzano in testa ad una lista le coordinate dei suoi 8-vicini neri;
2. si aggiornano delle variabili rappresentanti le coordinate massime e minime rispetto a quelle del pixel corrente;
3. si pone il valore del pixel corrente a un valore di comodo;
4. si estrae dalla lista il primo elemento e si ritorna al passo 1.

A questo punto si hanno le coordinate che delimitano ogni singola regione segmentata.
Per maggiori approfondimenti si veda, nella sezione 4, il paragrafo riguardante la funzione "**GrowingRegioni**".
Il secondo-blocco ha come scopo la pre-classificazione delle sotto-immagini estratte secondo la tipologia:

- segnale di obbligo
- segnale di pericolo
- segnale di indicazione
- regione di non interesse.

Nell'ultimo caso, ovviamente, sulla regione segmentata non verrà effettuata alcuna altra elaborazione, negli altri casi, invece, la sotto-immagine potrebbe essere, successivamente, sottoposta ad un classificatore neurale per individuare il tipo di informazione contenuta nel segnale.
La fase di classificazione si effettua mediante il calcolo di un coefficiente di similitudine tra le sotto-immagini estratte e le immagini "modello" rappresentanti i segnali stradali. Si tenga presente che il metodo presuppone che le due immagini da confrontare abbiano le stesse dimensioni e che le immagini tipo abbiano una dimensione fissata da ottenere mediante un apposito algoritmo di normalizzazione.
Un fattore da considerare è che la maggior parte delle volte l'acquisizione dell'immagine non avviene in asse rispetto al marker e, di conseguenza, le regioni candidate risultano ruotati: l'efficienza e la robustezza dell'algoritmo aumentano se il calcolo del fattore di correlazione viene effettuato non solo per l'immagine ottenuta dall'immagine reale ma anche le immagini ottenute da rotazioni attorno al punto centrale dell'immagine stessa.

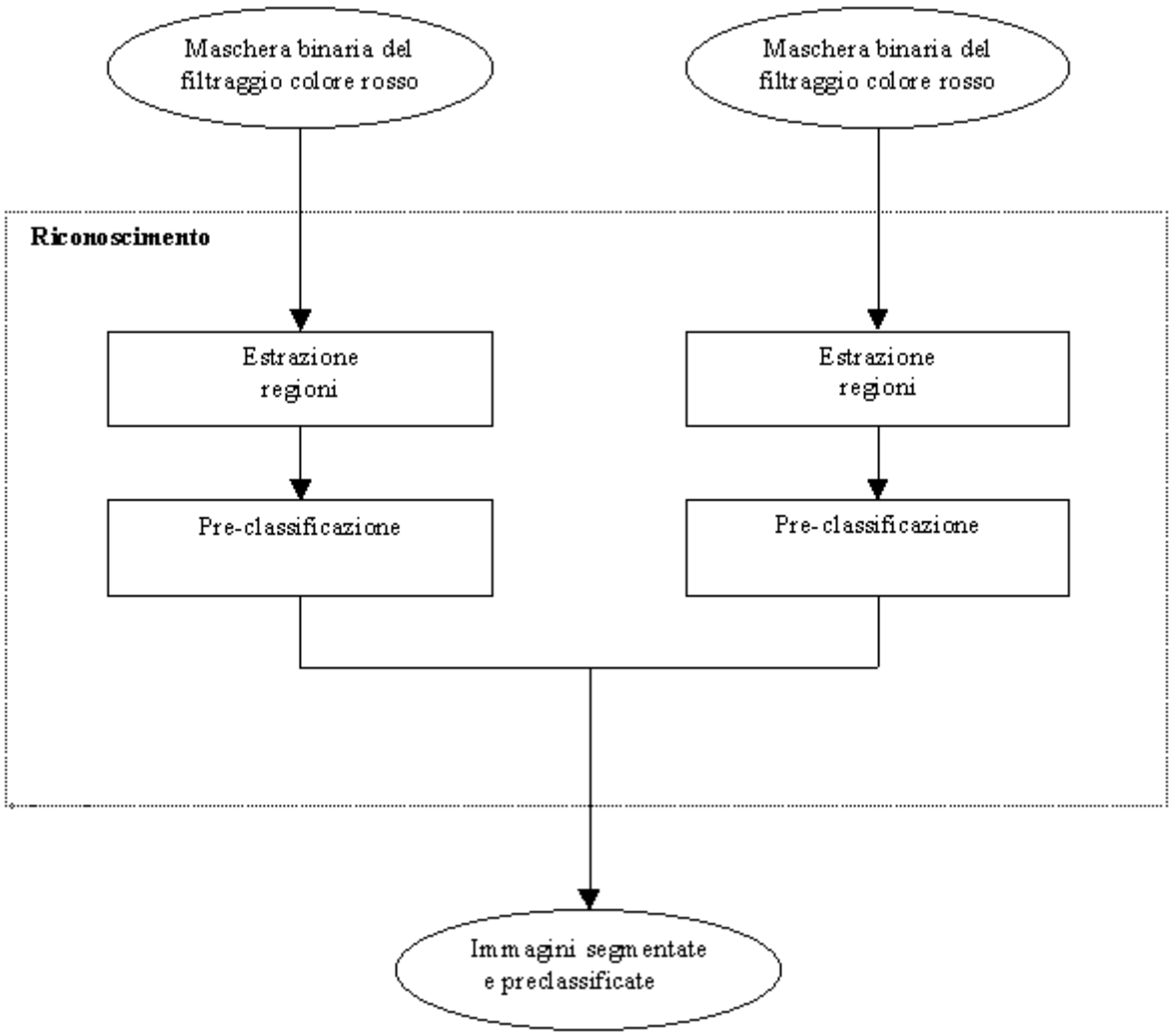


Fig 3. 6: Sotto-blocchi del blocco *riconoscimento*

In definitiva, l'algoritmo, da ripetere in modo assolutamente analogo, per il marker di forma triangolare e per quello di forma circolare è il seguente:

- generazione delle immagini di dimensione fissata rappresentanti i marker, da considerare come classe principale di appartenenza.
- per ogni regione individuata effettuare l'operazione di normalizzazione portandola alle dimensione fissata.
- effettuare delle rotazioni attorno al punto centrale, da un certo angolo negativo ad uno positivo secondo passi quantizzati.
- calcolare il fattore di similitudine tra l'immagine modello del marker e l'immagine modello e tra l'immagine modello e le sue rotazioni restituendo il massimo.

Mediante il calcolo del coefficiente di similitudine si determina il tipo del segnale stradale, in particolare:

- triangolo equilatero, segnale di pericolo

- triangolo equilatero verso il basso, segnale di “dare precedenza”
- circolare con bordo rosso, segnale di divieto
- circolare con sfondo blu, segnale di obbligo;

A questo punto la fase di pre-classificazione di ogni singola regione è completata.

Per maggiori approfondimenti si vedano, nella sezione 4, le sezioni riguardanti la funzione “**Normalizza**” e “**Correla**”.

4. Descrizione dei blocchi costituenti il sistema { $A(RS)^2$}

4.1 Introduzione

Nel presente capitolo si analizzano le singole parti facenti parte del sistema {$A(RS)^2$} ponendo particolare attenzione sulla procedura di parallelizzazione necessaria per effettuare il mapping su una macchina parallela, come SIMPil, dell’algoritmo seriale già esistente ed implementato in linguaggio C.

Nel primo paragrafo si analizza una funzione che implementa una particolare operazione necessaria in molti sotto-blocchi dei cui si è parlato nella sezione 3, alla quale ogni volta si farà riferimento esplicitamente; in ognuno dei paragrafi successivi si analizzerà una diversa funzione che implementa un sotto-blocco, come spiegato nella sezione precedente; la struttura delle funzioni di seguito descritte è assolutamente analoga a quella utilizzata nel sistema implementato in C.

4.2 Funzione Determinazione dei valori dei pixel adiacenti

Una situazione comune a molte delle funzioni descritte nei paragrafi seguenti è quella in cui si voglia confrontare il valore di una informazione relativa ad un pixel con quella degli 8 pixel adiacenti (a Nord-Ovest, Nord, Nord-Est, Est, Sud-Est, Sud, Sud-Ovest, Ovest).

Facendo riferimento all’architettura SIMPil, si tenga presente che il valore dei pixel è conservato in appositi registri (PR, Pixel Register) consecutivamente, quindi *linearmente*, mentre di seguito si farà riferimento ad una disposizione *matriciale* dei pixel, del tipo di Fig 4.1, in cui si suppone il

PPE (Pixel per Processor Element) pari a 16; il numero di ogni casella della matrice indica l'indice del pixel.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Fig 4. 1: Disposizione dei pixel all'interno dei PE quando il PPE è pari a 16

A questo punto, il problema della determinazione del valore dei pixel adiacenti si affronta con due diversi procedimenti:

- uno per i pixel adiacenti a NORD, SUD, OVEST, EST
- uno per i pixel adiacenti a NORD-OVEST, NORD-EST, SUD-OVEST, SUD-EST.

Supponendo sempre che il PPE sia pari a 16, con significato analogo a quello di figura precedente, in Fig 4.2 viene rappresentato il valore degli indici dei pixel appartenenti alla sotto-immagine presente in ogni processore (in nero) e quello dei pixel adiacenti nei processori adiacenti (a colori).

Si faccia attenzione al fatto che il pixel di cui si vuole determinare il valore, cioè il pixel adiacente, può trovarsi nello stesso processore oppure in un processore adiacente.

In riferimento al primo procedimento, si supponga di voler determinare il valore del pixel a NORD, per i casi SUD, OVEST, EST è assolutamente analogo.

Si riservi un registro per contenere il valore del pixel adiacente in entrambi i due casi:

1. il pixel a NORD fa parte dello stesso processore nel caso in cui il valore dell'indice del pixel sia uno dei seguenti: 4, 5, 6, 7 | 8, 9, 10, 11 | 12, 13, 14, 15.

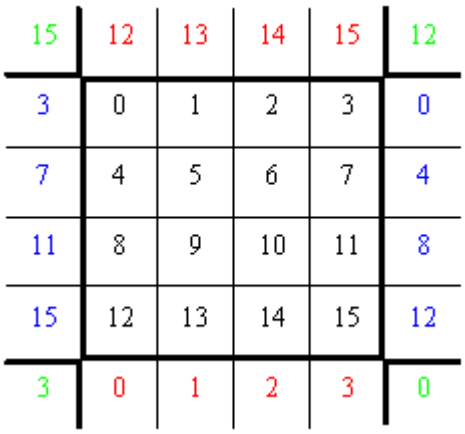


Fig 4. 2: Disposizione dei pixel all'interno dei PE e dei pixel adiacenti quando il PPE è pari a 16

2. il pixel a NORD fa parte dello stesso processore nel caso in cui il valore dell'indice del pixel sia uno dei seguenti: 0, 1, 2, 3.

La differenza tra i due casi dipende dal fatto che, secondo la rappresentazione di Fig 4.2, all'interno del singolo processore, il pixel sia a NORD oppure no.

I pixel a NORD sono 0, 1, 2, 3 e, secondo la rappresentazione binaria, utilizzando 16 cifre sono:

- 0 → 0000000000000000
- 1 → 0000000000000001
- 2 → 0000000000000010
- 3 → 0000000000000011

quindi sono del tipo:

000000000000000XX

Detto ciò per determinare le informazioni necessarie relative al pixel a NORD, le operazioni da effettuare sono:

1. si inserisca il valore dell'indice del pixel in un registro.
2. si effettuino due shift aritmetici a destra.
3. si confronti il valore con 0 e, per vedere se l'indice del pixel è 0, 1, 2, 3 si distinguendo opportunamente i due casi per mezzo dell'apposita istruzione SLEEP, facente parte del set di istruzioni di SIMPil:

a. il pixel a NORD si trova nello stesso processore e il suo indice si ottiene sottraendo 4 al valore dell'indice del pixel attuale (IP) (per esempio, se IP = 4 allora $IP_{NORD} = 0$).

b. il pixel a NORD si trova nel processore a NORD e il suo indice si ottiene sommando 12 all'indice del pixel attuale (per esempio, se IP = 0 allora $IP_{NORD} = 12$) e prelevando, con l'apposita istruzione XFER, facente parte del set di istruzioni di SIMPil, il valore del processore a NORD.

4. Si pone nel registro di destinazione il valore dell'indice del pixel a NORD.

Per quanto riguarda l'analisi del pixel a SUD, ad OVEST e ad EST, le fasi da effettuare sono assolutamente analoghe.

In riferimento al secondo procedimento si supponga di volere determinare il valore del pixel a NORD-OVEST, per i casi SUD-OVEST, NORD-EST, SUD-EST, è assolutamente analogo.

Si riservi un registro per contenere il valore del pixel adiacente in ognuno dei quattro casi:

1. il pixel a NORD-OVEST fa parte dello stesso processore nel caso in cui il valore dell'indice del pixel sia uno dei seguenti: 5, 6, 7 | 9, 10, 11 | 13, 14, 15.
2. il pixel a NORD-OVEST fa parte del processore a NORD nel caso in cui il valore dell'indice del pixel sia uno dei seguenti: 1, 2, 3.
3. il pixel a NORD-OVEST fa parte del processore a OVEST nel caso in cui il valore dell'indice del pixel sia uno dei seguenti: 4, 8, 12.
4. il pixel a NORD-OVEST fa parte del processore a NORD-OVEST nel caso in cui il valore dell'indice del pixel sia 0.

In modo simile al procedimento riguardante la determinazione del valore del pixel a NORD, in questo caso, le quattro situazioni differiscono a seconda del fatto che, sempre in riferimento alla Fig 4.2, all'interno del singolo processore, il pixel a NORD-OVEST si trovi all'interno dello stesso processore, del processore a NORD, del processore a OVEST, del processore a NORD-OVEST.

I pixel per i quali il pixel a NORD-OVEST si trovi nel processore a NORD sono 1, 2, 3 e, secondo la rappresentazione binaria sono:

- 1 $\rightarrow$ 0000000000000001
- 2 $\rightarrow$ 0000000000000010
- 3 $\rightarrow$ 0000000000000011

sono tutti quelli del tipo:

000000000000000XX

eccetto

0000000000000000.

I pixel per i quali il pixel a NORD-OVEST si trovi nel processore a OVEST sono 4, 8, 12 e, secondo la rappresentazione binaria sono:

- 4 → 0000000000000100
- 8 → 0000000000001000
- 12 → 0000000000001100

sono tutti quelli del tipo:

000000000000XX00

eccetto

0000000000000000.

Il pixel per il quale il pixel a NORD-OVEST si trovi nel processore a NORD-OVEST è 0 che secondo la rappresentazione binaria è 0000000000000000.

Le operazioni da effettuare sono:

1. si inserisca il valore dell'indice del pixel in un registro.
2. si controlli se l'indice del pixel è del tipo 000000000000000XX, come nel caso del procedimento relativo alla determinazione del valore del pixel a NORD, ma non sia 0000000000000000. In questo caso l'indice del pixel a NORD-OVEST si ottiene sommando 11 al valore del pixel attuale (per esempio se IP = 1 allora $IP_{NORD\text{-}OVEST}$ = 12) e si preleva, con l'apposita funzione XFER, il valore del processore a NORD.
3. si controlli se l'indice del pixel è del tipo 000000000000XX00 ma non sia 0000000000000000: si effettuano quattro shift aritmetici a destrae si confronta il contenuto con 0, poi si riporta il contenuto nella postazione iniziale, si effettuano 14 shift aritmetici a sinistra e si confronta il contenuto con 0, in questo modo si controlla se gli ultimi due bit sono pari a 0. In questo caso l'indice del pixel a NORD-OVEST si ottiene sottraendo 1 al

valore del pixel attuale (per esempio se IP = 4 allora $IP_{NORD-OVEST}$ = 3) e si preleva, con l'apposita funzione XFER, il valore del processore a OVEST.

4. se il valore dell'indice non è del tipo 00000000000000XX né 000000000000XX00, l'indice del pixel a NORD-OVEST si ottiene sottraendo 5 all'indice del pixel attuale (per esempio se IP = 5 allora $IP_{NORD-OVEST}$ = 0).
5. se il valore dell'indice è 0000000000000000, allora il pixel a NORD-OVEST si trova nel processore a NORD-OVEST e il suo indice si ottiene sommando 15 all'indice del pixel attuale (IP = 0, $IP_{NORD-OVEST}$ = 15, IP_{NORD} = 12, IP_{OVEST} = 3), quindi si preleva, con l'apposita istruzione XFER, il valore del processore a NORD (IP = 0, IP_{NORD} = 12 e IP_{OVEST} = 3, $IP_{NORD-OVEST}$ = 15) e si preleva dal processore a OVEST (IP = 0, IP_{OVEST} = 3 ma IP_{OVEST} = 3 contiene $IP_{NORD-OVEST}$ = 0)
6. Si noti che la distinzione dei quattro casi viene opportunamente gestita mediante il corretto uso della istruzione SLEEP.
7. si pone nel registro R_y il valore dell'indice del pixel a NORD-OVEST

4.3 Funzione Filtro SNN

La funzione, come già detto nel paragrafo 3.2.1, ha lo scopo di ridurre il rumore puntuale che può presentarsi in seguito all'acquisizione dell'immagine; le operazione necessarie consistono, semplicemente, nell'implementazione delle formule riportate a pag. 9, le quali suppongono l'utilizzo di una funzione che implementi l'operazione di Radice Quadrata.

L'unico problema che presenta la parallelizzazione della funzione è quello della determinazione dei valori dei pixel adiacenti, i quali possono trovarsi in processori adiacenti: il problema si risolve per mezzo della funzione descritta nel paragrafo 4.1.

4.4 Funzione Conversione RGB→HSV

Di tale funzione esiste già un'implementazione su SIMPil, riportata nella sezione 5, i cui risultati sono stati testati con successo.

La funzione ha, semplicemente, lo scopo di implementare le formule di conversione dallo spazio dei colori RGB allo spazio dei colori HSV**:**

V=max {R,G,B}

$$S = 1 - \left[\frac{3}{R+G+B}\right]a$$

essendo a=min {R,G,B}

$$H = \cos^{-1}\left[\frac{0.5((R+G)+(R-B))}{\sqrt{(R-G)^2+(R-B)(G-B)}}\right]$$

Se S=0 allora H non è definita.

Si vede che, specie per la componente H, la formula di conversione richiede un notevole costo computazionale specie se, come in questo caso il mapping deve essere effettuato su una macchina ad aritmetica intera.

Si possono utilizzare delle formule più semplici che, seppur approssimate, conducono a risultati soddisfacenti.

La conversione eseguita in questo lavoro fa riferimento alla formule di Smith:

max = massimo{ R, G, B}

min = minimo { R, G, B}

V=max {R,G,B}

$$S = \frac{\max - \min}{\max}$$

$$H = \begin{cases} \left(\frac{(G - B)}{\max - \min}\right) * 60 & \text{se } R = \max \\ 2 + \frac{(B - R)}{\max - \min} * 60 & \text{se } G = \max \\ 4 + \frac{(R - G)}{\max - \min} * 60 & \text{se } B = \max \end{cases}$$

Si utilizza un ciclo avente un numero di iterate pari al PPE, in cui l'iterata i-ma fa riferimento al pixel i-mo; all'interno di ogni iterata, per determinare le coordinate H, S, V del pixel i-mo, è sufficiente conoscere il valore delle coordinate R, G, B del pixel i-mo, senza alcun confronto con informazioni relative ad altri pixel: non è necessaria alcuna comunicazione tra i PE e quindi l'algoritmo segue le stesse fasi dell'algoritmo seriale senza particolari operazioni di parallelizzazione.

*4.5 Funzioni **Filtro nello spazio HSV Rosso** e **Filtro nello spazio HSV Blu***

Le due funzioni sono assolutamente analoghe, per semplicità, si farà riferimento alla prima e, quanto detto, vale anche per la seconda.

La funzione *Filtro nello spazio HSV Rosso* serve a creare una maschera binaria, cioè una matrice il cui numero delle righe e delle colonne è pari a quello dell'immagine di partenza.

Il valore di ogni elemento della matrice è 0 o 1 a seconda del fatto che, nell'immagine di partenza, il pixel corrispondente, secondo particolari valori di soglia di H, S, V fissati per le tre componenti, non sia classificabile come "rosso" o meno.

La creazione della maschera avviene parallelamente all'interno di ognuno dei processori paralleli, indipendentemente l'uno dall'altro, ognuno in relazione alla sotto-immagine presente nel processore stesso. Poiché l'analisi sul singolo elemento della maschera dipende soltanto dal valore del corrispondente pixel nella immagine di partenza e, quindi, non è necessaria comunicazione alcuna tra i processori, l'algoritmo seriale, già implementato in C, non necessita di operazioni particolari di parallelizzazione.

Le operazioni da effettuare sono:

- Si esegue un ciclo avente un numero di iterate uguale al numero di pixel presenti all'interno di ogni processore.

- All’interno di ogni iterata:
    1. si controlla per l’i-mo pixel che:
        a. il valore di H sia compreso tra due valori di soglia fissati.
        b. il valore di S sia maggiore o uguale del valore di soglia fissato.
        c. il valore di V sia maggiore o uguale del valore di soglia fissato.
    2. vengono posti nello stato di SLEEP tutti quei processori il cui pixel i-mo non soddisfi almeno un controllo.
    3. se tutti i controlli sono stati superati e, quindi, il pixel i-mo può essere considerato come rosso, si pone 1 nella corrispondente posizione all’interno della maschera binaria.
    4. tutti i processori attualmente nello stato attivo, vengono posti nello stato di SLEEP e si riattivano tutti gli altri processori che non hanno superato almeno uno dei controlli.
    5. viene posto 0 nella posizione all’interno della maschera binaria.
- si riattivano tutti i processori.

*4.6 Funzioni **Filtro neri isolati** e **Filtro bianchi isolati***

Le due funzioni sono assolutamente analoghe, per semplicità, si farà riferimento alla prima e, quanto detto, vale anche per la seconda.

Lo scopo delle due funzioni è quello di eliminare i disturbi puntuali all’interno della maschera binaria appena costruita; in riferimento alla prima, infatti all’interno della maschera binaria, un pixel nero (cioè il cui valore è 1) circondato da soli pixel bianchi (cioè il cui valore è 0), probabilmente anche il pixel nero dovrebbe essere bianco: la funzione rende bianchi tutti quei pixel neri che hanno almeno 6 degli 8 pixel adiacenti.

Poiché, quindi, si deve controllare il valore dei pixel adiacenti è necessario, sicuramente, una comunicazione dati tra processori, quindi è necessaria una parallelizzazione dell’algoritmo già esistente: è vero, cioè, che il carico di lavoro viene diviso all’interno dei processori, e quindi il tempo di esecuzione viene ridotto, però è necessario anche del tempo relativo allo scambio dati tra processori che si ottiene utilizzando la funzione analizzata nel paragrafo 4.2 relativamente alla determinazione del valore del pixel adiacente.

Le operazioni da effettuare sono:

- Si inizializza a 0 la variabile CONT che conta il numero di pixel adiacenti trovati pari a 0.
- si esegue un ciclo avente un numero di iterate pari al numero di pixel contenuti all’interno di ogni processore (il PPE), ogni iterata si riferisce ad un pixel.

- se il pixel relativo i-mo, non è nero, cioè se all'interno della maschera binaria costruita il valore trovato non è pari a 1:

  allora il processore viene posto in uno stato di SLEEP per l'intera iterata.

  altrimenti viene eseguita un'iterata

- all'interno di ogni iterata l'iterata:
  1. si determina il valore dei pixel adiacenti chiamando la funzione "determinazione pixel adiacente" per il pixel a NORD-OVEST.
  2. se è pari a 0 si incrementa il contatore CONT.
  3. si effettuano i passi 1. e 2. per gli altri sette pixel adiacenti
  4. se il contenuto del contatore è ≥ 6, cioè almeno 6 dei pixel adiacenti sono bianchi *allora*all'interno della maschera binaria, si pone il 2.
- si esegue un secondo ciclo nel quale vengono poste uguali a 0 tutte le locazioni trovate pari a 2 all'interno della maschera binaria.

*4.7 Funzioni **Seme Rosso** e **Seme Blu***

Le due funzioni sono assolutamente analoghe, per semplicità, si farà riferimento alla prima e, quanto detto, vale anche per la seconda.

La funzione *Seme Rosso* ha lo scopo di risolvere il primo problema del *region growing* (si veda quanto detto nel paragrafo 3.2.2), cioè quello di ricavare le componenti H, S, V del seme usato per il growing.

Analogamente al caso del paragrafo 4.4, la determinazione del pixel seme avviene parallelamente all'interno di ognuno dei processori paralleli, indipendentemente l'uno dall'altro: l'analisi sul singolo elemento della maschera dipende soltanto dal valore del corrispondente pixel nella immagine di partenza e, quindi, non è necessaria comunicazione alcuna tra i processori, l'algoritmo seriale, già implementato in C, non necessita di operazioni particolari di parallelizzazione.

Le operazioni da effettuare sono:

- si inizializzano a 0 le tre variabili H_SEME, S_SEME, V_SEME che contengono i valori H, S, V del seme e la variabile PUNTEGGI che conta il numero di pixel rossi
- si esegue un ciclo avente un numero di iterate pari al numero di pixel contenuti nel processore.
- All'interno di ogni iterata, il pixel i-mo all'interno della matrice binaria MASCHERA è pari a 1, cioè il pixel corrispondente è rosso;
  1. viene incrementata la variabile PUNTEGGI.

2. i valori H_SEME, S_SEME, V_SEME vengono incrementati dei valori di H, S, V del pixel i-mo, tenendo presente che, se il valore di H è >330, viene decrementato di 360 e poi sommato.

- se il numero di pixel rossi trovati, di cui tiene traccia la variabile PUNTEGGI, è minore di una certa percentuale

  allora i tre valori H_SEME, S_SEME e V_SEME vengono posti a –1

  Altrimenti i valori H, S, V del seme si ottengono come media dei valori H, S, V dei pixel trovati rossi cioè si ottengono dividendo i valori attuali delle tre variabili per la variabile PUNTEGGI; a tal fine è necessario implementare un algoritmo per la funzione *Divisione*.

*4.8 Funzione **Growing***

Tale funzione esegue il secondo step del processo del *region growing*, cioè quello di aggregare i pixel, a partire dalla conoscenza delle coordinate H, S, V del seme.

Nell'algoritmo seriale vengono aggregati tutti i pixel che rispettano il principio di similitudine, all'interno di ogni sotto-regione, senza alcuna comunicazione tra le sottoregioni: nell'algoritmo parallelo non vi sono sostanziali differenze di implementazione rispetto a quello già esistente, è sufficiente effettuare le stesse operazioni tenendo presente che, in questa versione, l'intero procedimento viene effettuato in parallelo all'intermo di ogni processore.

Le operazioni da effettuare sono le seguenti:

- si esegue, con un ciclo che inizializza i valori di due matrici aventi la stessa disposizione righe-colonne della matrice di pixel contenente la sotto-immagine del processore stesso corrispondente alla sotto-regione:
  a. la matrice binaria MASCHERA, che tiene traccia del fatto che il pixel cui si riferisce all'interno della iterata faccia parte o meno della regione che si sta costruendo e che viene inizializzata a 0.
  b. la matrice GROW, la quale tiene traccia del fatto che il pixel sia stato già considerato o meno all'interno del processo di aggregazione e ogni suo valore viene inizializzato a 255.
- si esegue un ciclo avente un numero di iterate pari al numero di pixel contenuti in ogni processore.
- all'interno di ogni iterata:
  1. si determina il grado di similitudine tra il pixel i-mo ed il pixel seme; utilizzando l'apposita istruzione SLEEP, si distinguono due casi:

se:

a. la percentuale di pixel rossi all'interno della sotto-regione è superiore a quella minima
b. il valore di S del pixel i-mo è superiore al valore soglia
c. il valore di V del pixel i-mo è superiore al valore soglia

allora Per i processori ancora attivi, si determina il grado di similitudine,di cui tiene conto la variabile DISTANZA, si determina tramite la funzione *DistanzaHSV,* la quale riceve in ingresso le coordinate H, S, V del pixel i-mo e quelle del pixel seme e ne calcola il grado di similitudine; per implementare tale funzione è necessario l'utilizzo della funzione *Radice Quadrata* e della funzione da implementare *Coseno*.

altrimenti si pone il valore di DISTANZA pari a 0.

2. Si effettua un altro controllo multiplo, mantenendo attivi soltanto i processori con l'utilizzo dell'istruzione SLEEP, per i quali si verifichi contemporaneamente che:
   a. il valore del pixel i-mo di H sia ≠ -1
   b. il valore del pixel i-mo di S e superi il valore soglia
   c. il valore del pixel i-mo di V e superi il valore soglia
   d. il valore determinato per la DISTANZA superi il valore soglia prefissato.
3. Si costruisce una lista contenente per primo l'indice del pixel i-mo, poi quello di tutti i pixel facenti parte della particolare regione che si sta aggregando all'interno della sotto-immagine. Su SIMPil si può implementare la lista inserendo i valori degli indici dei pixel in celle di memorie contigue, a partire dalla locazione di memoria etichettata, per esempio, con la label LISTA, ed utilizzando un registro del controllore come contatore per tenere traccia del numero di elementi della lista e di conseguenza del numero di locazioni che, all'interno della memoria, fanno parte della lista: l'incremento del contatore, rappresentato dalla variabile NUM_LISTA,, indica l'inserimento di un elemento all'interno della lista. La costruzione della lista avviene per mezzo della funzione *Espandi Successori*, di seguito descritta, all'interno della quale si incrementa la variabile NUM_LISTA ogni volta che si inserisce un elemento nella lista.
4. Quando la lista è stata interamente costruita, a partire dalla locazione di memoria etichettata con la label LISTA, contenente i valori degli indici dei pixel che costituiscono la regione aggregata, in numero pari al valore del contatore, con un ciclo, si scorre nuovamente la lista e, in corrispondenza ai valori degli indici contenuti nella lista, si modificano i valori della matrice MASCHERA, da 0 a 1, per indicare che tali pixel fanno parte della regione

aggregata. Per quanto riguarda il numero di iterate del ciclo, si tenga presente che, considerato che l'esecuzione avviene parallelamente all'interno di ogni processore e che il controllore è, invece, unico, per tutti i processori il numero di iterate deve essere pari al numero di elementi della lista avente il massimo numero di elementi; tutti i processori aventi nella lista un numero di elementi minore di quello fissato, vengono posti nello stato di SLEEP, nelle iterate necessarie a completare lo scorrimento delle liste più lunghe, e riattivati quando si esauriscono gli elementi da analizzare. Il numero di elementi massimo che una lista può contenere è pari, al limite, quando la regione che si sta costruendo è costituita dall'intera sotto-immagine, al numero totale di pixel facenti parte del processore stesso: è questo il numero di iterate necessario e che si conta per mezzo di un registro del controllore che funge proprio da contatore, deve essere effettuato per scorrere l'intera lista.

5. si inizia l'iterata successiva relativa all'analisi del pixel i+1-mo.

Si ripongono nello stato attivo tutti i processori per i quali non viene effettuata la aggregazione dei pixel perché non verificavano le condizioni del test di ingresso.

*4.8.1 Funzione **Espandi Successori***

Come già detto, tale funzione serve per costruire la lista contenente tutti i pixel facenti parte della particolare regione da aggregare.

La funzione *Espandi Successori* viene eseguita all'interno della iterata i-ma del ciclo principale della funzione *Growing* relativa al pixel i-mo; il valore "i" del pixel viene inserito come primo elemento della lista.

All'interno di tale funzione:

- si scorre la lista che si sta costruendo, inizialmente costituita dal solo indice i del pixel i-mo
- per ogni elemento si richiama la funzione *Espandi Successori* la quale effettua un controllo sui pixel adiacenti al pixel contenuto nell'elemento della lista per vedere se inserirli o meno all'interno della lista ed analizzarli in un secondo tempo e richiamando, anche per questi, la funzione *Espandi Successori*.

Per scorrere gli elementi della lista si utilizza un registro che punta via via all'elemento della lista da espandere e, poiché l'inserimento all'interno della lista viene effettuato in coda, si utilizza un registro di supporto che punta all'ultimo elemento della lista, cioè all'ultima locazione di memoria riservata per la lista.

Le operazioni da effettuare sono:

- si esegue un ciclo mediante il quale si controlla che il pixel i-mo abbia un valore di H ≠ -1: se sì, si ponga a 0 il valore del pixel i-mo all'interno della matrice GROW e si pongano nello stato di SLEEP tutti i processori per i quali si verifica ciò, e vengono riattivati soltanto al termine dell'esecuzione della funzione stessa.
- si controlla se gli 8 pixel adiacenti, a NORD-OVEST, NORD, NORD-EST, EST, SUD-EST, SUD, SUD-OVEST, OVEST, possano essere aggregati o meno: il tipo di controllo che si deve effettuare è sempre lo stesso, ciò che cambia è soltanto il pixel su cui effettuare il controllo, quindi si pone in un registro il valore dell'indice del pixel adiacente, negli otto diversi casi e poi si richiama la funzione *Confronto Adiacente* la quale opera, di volta in volta, sul contenuto del registro.
- Per la determinazione del pixel a NORD-OVEST:
  1. se non fa parte della sotto-immagine relativa al processore (cioè se non ha configurazione del tipo: 00000000000000 XX e/o 000000000000XX00, si veda a tal proposito, il paragrafo 4.1):
     allora se si sta analizzando come pixel a NORD-OVEST un pixel avente tale configurazione, si pone il processore nello stato di SLEEP.
     altrimenti il pixel a NORD-OVEST ha un indice pari all'indice attuale –5
  2. si inserisce il valore determinato per l'indice in un apposito registro e si richiama la funzione *Confronto Adiacente*.
- Analogo discorso per il confronto con gli altri sette pixel adiacenti.

*4.8.2 Funzione **Confronto Adiacente***

Lo scopo di questa funzione è quello di determinare se inserire o meno all'interno della lista di aggregazione il pixel adiacente in esame.
Le operazione da effettuare sono:

- si controlla all'interno della matrice GROW il valore del pixel, il cui indice è quello contenuto nella lista
  se è pari a 255, cioè non è stato modificato
  allora si pongono nello stato di SLEEP tutti i processori per i quali questa condizione è verificata al fine di verificare che il pixel che si sta analizzando non sia stato già considerato; infatti, come si vedrà, ogni volta che si analizza

un pixel, il corrispondente valore all'interno della matrice GROW, viene posto = a 0.

- Per i processori ancora attivi, se:
  a. il valore di S del pixel adiacente supera il valore soglia
  b. il valore di V del pixel adiacente supera il valore soglia
  c. richiamando la funzione *distanza_hsv*, che il grado di similitudine tra il pixel adiacente ed il pixel attuale superi il valore di soglia.

  allora
  1. si pone in lista l'indice del pixel adiacente.
  2. si incrementa il contatore del numero di elementi della lista.
  3. nella locazione di memoria puntata dal registro di comodo, viene inserito il valore dell'indice del pixel adiacente.
  4. si pone 0 all'interno della matrice GROW, in corrispondenza dell'indice determinato.

*4.9 Funzione **Growing Regioni***

Questa funzione ha come obiettivo la individuazione e la estrazione delle regioni di interesse dalla immagine di partenza.

A partire dalle regioni in cui i valori della matrice MASCHERA sono pari a 1, all'interno della intera immagine, si deve individuare la regione di interesse e il relativo rettangolo che la circoscrive al fine di determinare il numero di righe e di colonne di pixel che costituiscono la regione e il fattore di scala rispetto al numero di righe e di colonne totali.
Si deve effettuare un'analisi sull'intera immagine di partenza e non soltanto sulla singola sotto-immagine appartenente al processore, quindi la parallelizzazione è più complessa rispetto ai casi precedenti.
Si supponga che la matrice di processori di SIMPil sia costituita da 16 processori (4x4) come in Fig 4.3. Nell'algoritmo già implementato, le varie operazioni vengono effettuate serialmente sull'immagine complessiva, qui, invece, l'intera immagine è suddivisa all'interno dei vari processori, quindi l'operazione mediante la quale si circoscrive una regione individuata viene fatta in parallelo sui vari processori.
Nell'algoritmo seriale si utilizza la matrice MASCHERA, determinata come scritto nel paragrafo 4.2 e una matrice GROW i cui valori vengono inizializzati, con un ciclo, a 255 e la cui utilità è analoga a quella della omonima matrice utilizzata nel paragrafo 4.2.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Fig 4. 3: Matrice di processori SIMPil

Analogamente al caso della costruzione della lista relativa alla funzione *Growing*, con un ciclo che analizza tutti i pixel all'interno di ogni PE, si costruiscono delle liste contenenti i pixel che costituiscono le regioni da circoscrivere; il criterio utilizzato per inserire l'indice di un pixel in lista si basa sul fatto che il pixel faccia parte della regione, e quindi il valore corrispondente della matrice MASCHERA sia pari a 1, e che non sia stato già considerato, e quindi il valore corrispondente nella matrice GROW deve essere pari a 255.
Questo meccanismo è quello già implementato nell'algoritmo esistente; in realtà, invece di costruire due liste per ogni regione, aventi gli stessi contenuti, si dovrebbe modificare l'algoritmo ed eseguire tutte le operazioni, di seguito descritte, all'interno delle liste costruite nel paragrafo precedente, invece che all'interno del secondo tipo di liste, così da non ripetere due volte le stesse operazioni.

All'interno del ciclo tutti i processori parallelamente, ma serialmente all'interno di ogni processore, si costruisce una lista contenente i pixel che costituiscono una regione, che è in realtà una sotto-regione della regione complessiva, suddivisa tra i vari pixel: cioè se, all'interno del processore si ha un'immagine del tipo di Fig 4.4 a, nel ciclo corrispondente alla prima regione, è come se si stesse costruendo solo la prima regione, come si vede in Fig 4.4 b.

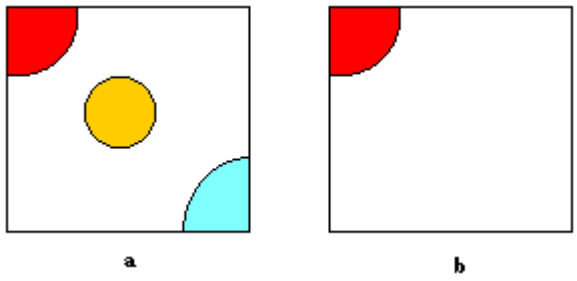


Fig 4. 4: Immagine di partenza e immagine costruita dopo un'iterata

Si tenga presente che nella lista è presente l'indice del pixel, supponendo che gli indice vengano memorizzati serialmente, mentre l'analisi dell'immagine si effettua a partire da una matrice bidimensionale: si deve trasformare l'indice del pixel da seriale a bidimensionale.

Supponendo che il numero di pixel all'interno di ogni processore (il PPE) sia pari a 16, la matrice bidimensionale è costituita da 4 righe (0,1,2,3) e 4 colonne (0,1,2,3), come in Fig 4.5, nella quale il contenuto della matrice indica l'indice del pixel.

	0	1	2	3
0	0	1	2	3
1	4	5	6	7
2	8	9	10	11
3	12	13	14	15

Fig 4. 5: Indici della matrice bidimensionale contenente gli indici

L'indice riga (IR) dell'indice (I) del pixel si ottiene effettuando la divisione intera di questo per il numero di pixel lungo la riga, l'indice colonna (IC) si ottiene come resto della precedente divisione. Per esempio il pixel con indice I = 6, calcolando l'IR e l'indice colonna IC, si determina che:

$$IR = 6/4=1$$

$$IC = 6\%4=2$$

che, come si può vedere in Fig 4.5 sono i valori corretti.

Per determinare il rettangolo che circoscrive la sotto-regione, all'interno di ogni processore, è sufficiente determinare i valori minimi e massimi dell'indice riga e dell'indice colonna poiché gli estremi in alto e a sinistra e in basso e a destra del rettangolo hanno come coordinate, rispettivamente, minimo indice riga - minimo indice colonna e massimo indice riga - massimo indice colonna.

Poiché all'interno di ogni ciclo è possibile individuare più di una sotto-regione, in genere servono più di quattro variabili: per la precisione, sono necessarie 4 variabili per ogni sotto-regione che si

determina, quindi tutte le variabili che si riferiscono ai rettangoli circoscriventi vengono salvate in memoria a partire da una locazione di memoria etichettata, per esempio, con la label ESTREMI. All'interno del ciclo è necessaria anche una variabile NUM_REGIONI, che conta il numero di aree e che viene incrementata ogni volta che viene individuata una nuova area e, quindi, costruita una nuova lista: in totale, quindi, le locazioni di memoria necessarie per memorizzare le coordinate relative ai rettangoli circoscriventi sono: 4* NUM_REGIONI.
Per esempio si faccia riferimento alla Fig 4.16 l'immagine di partenza è un cerchio, questa però viene suddivisa in 4 processori, in ognuno dei quali vengono conservate le parti colorate, in giallo, azzurro, rosa e verde; in ogni processore si effettua in

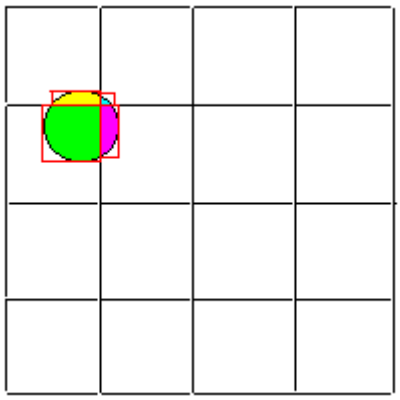


Fig 4. 6: Immagine di partenza con rettangoli che circoscrivono le sotto-regioni

parallelo, l'operazione di circoscrizione (tratto in rosso) per mezzo di un rettangolo delle sotto-immagini immagazzinate; il problema è che la composizione dei rettangoli che iscrivono le sotto-immagini, trascurando i lati di confine tra i processori, non corrisponde al rettangolo che circoscrive la composizione delle sotto-figure, cioè la figura di partenza (tratto in rosso di Fig 4.7), quindi si deve ricostruire i rettangolo di partenza.

Si tenga presente, inoltre, che quanto detto vale nel caso in cui ogni PE contenga soltanto una sotto-immagine, negli altri casi, invece, più sotto-immagini, relative a diverse immagini di base, la determinazione del rettangolo che circoscrive le sotto-immagini appartenenti allo stesso PE si effettuano serialmente all'interno di ogni PE, ma parallelamente nei diversi PE.

E' stato detto che, a partire dai rettangoli che circoscrivono la sotto-figura interna ad ogni processore, si deve costruire il rettangolo che circoscrive l'intera figura, e determinare il numero di righe e di colonne della regione determinata: questo deve essere effettuato per ognuna delle sotto-regioni interne al processore, quindi, le fasi di seguito descritte devono essere effettuate, all'interno di un ciclo, un numero di volte pari al valore della variabile NUM_REGIONI interno ad ogni

processore; poiché, sebbene il carico di lavoro sia distribuito tra i vari processori, il controllore è unico il numero di iterate all'interno del ciclo deve essere pari al numero di aree possibile, cioè al PPE, visto che, al limite, un'area può essere costituita da un solo pixel e che tutti i pixel possono costituire una regione.

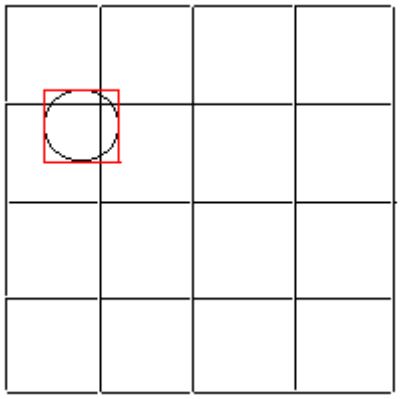


Fig 4. 7: Immagine di partenza con rettangolo che circoscrive l'intera regione

Chiaramente, questa ipotesi si può facilmente ridurre, per esempio, considerando trascurabile un'area costituita da un solo pixel e supponendo di considerare soltanto aree aventi X pixel, con X da fissare opportunamente: il numero necessario di iterate da PPE viene ridotto a PPE/X.
In ogni iterata:

- si scorre la memoria, a partire dalla locazione ESTREMI, per considerare la singola sotto-regione.
- per ogni regione individuata, si deve vedere se questa è interamente contenuta all'interno del processore oppure se è una sotto-regione estesa tra più processori. Per fare ciò, basta controllare se gli ESTREMI siano valori di confine: se la sotto-immagine interna ad un processore è del tipo di Fig 4.8 allora il valore minimo per le righe (minRow) è pari a 0.

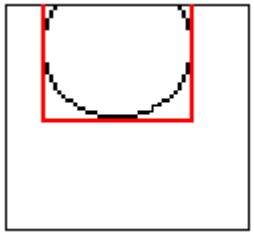


Fig 4. 8: Regione appartenente anche al PE a NORD

quindi:

$$minRow = 0 \quad (1)$$

Analogamente, per le altre tre situazioni si verifica che:

$$maxRow = NumRighe-1 \quad (2)$$

$$minCol = 0 \quad (3)$$

$$maxCol = NumCol-1 \quad (4)$$

dove maxRow indica il numero più grande dell'indice riga, minCol e maxCol il numero più piccolo e più grande dell'indice colonna, NumRighe e NumCol indicano il numero di righe e di colonne di pixel conservati all'interno di ogni processore.

Se nessuna delle quattro condizioni è verificata, allora la regione non è suddivisa tra i vari processori; se, invece, minRow=0, per esempio, è possibile che la regione individuata sia la sotto-regione di una regione che si estende nel processore a NORD: si deve quindi verificare se nel processore a NORD, per almeno n'area si verifichi che maxRow = NumRighe –1.

Si effettua, quindi, un ciclo per mezzo del quale si determina il valore di maxRow del PE a NORD e che termina quando si verifica una delle 2:

1. maxRow=0
2. sono stati controllati i valori maxRow di tutte le regioni nel PE a NORD.

Supposto che si verifichi che minRow=0 e $MaxRow_{NORD}$=NumRighe-1, si deve ancora effettuare un controllo sui valori di min minCol e maxCol, per verificare che la regione sia una sotto-regione di una regione più ampia.

Se si verifica una delle seguenti di Fig 4.9 le due sottoregioni sono separate.

Fig 4. 9: Sotto-regioni separate

Se si verifica una delle seguenti di Fig 4.10 le regioni che si individuano non sono di interesse poiché le regioni che si devono individuare hanno forma triangolare o circolare;

Fig 4. 10: Sotto-regioni non circolari e non triangolari

quindi si deve effettuare un controllo dei valori di minCol e di maxCol del processore e del processore a NORD e verificare che la situazione sia una di quelle di Fig 4.11.

Fig 4. 11: Possibili sotto-regioni di interesse

Più schematicamente: nel listato C, per fare il Growing Regioni, si utilizza la matrice GROW, che tiene conto dei pixel considerati, si controlla nella matrice MASCHERA se il pixel è stato analizzato oppure no e, a partire da questi controlli si costruisce la lista, si scorrono gli elementi della lista e si determinano i quattro valori di minRow, maxRow, minCol, maxCol. Nel seguito è riportato l'algoritmo per determinare, all'interno del singolo PE, minRox, maxRow, minCol, maxCol.

Algoritmo per la determinazione delle variabili minRox, maxRow, minCol, maxCol.

Sommariamente, per il confronto con il pixel a SUD, l'algoritmo è:

1. area già considerata?
2. No: il PE è un PE a SUD?
3. No: dalla locazione di memoria ESTREMI si considera il valore di maxRow;

   maxRow = NumRighe in un PE - 1?
4. Sì: LOOP con numero di iterate = NUM_REGIONI:

   la situazione è del tipo di Fig 4.17 ?:
5. Sì: LOOP con un numero di iterate pari al NUM_REGIONI del PE a SUD

   MinRow del PE a SUD = 0 ?:

   Sì: momentaneamente il PE viene posto nello stato di SLEEP

   No:

   a. ESTREMI = ESTREMI + 4
   b. il numero di aree da analizzare nel PE a SUD è decrementato
   c. il numero di aree da analizzare nel PE a SUD ≥ 0?

Sì: ripeti iterate

6. i processori nello stato di SLEEP vengono riattivati:
7. si pone: minCol = min {minCol, minCol a SUD}

   maxCol = max {maxCol, maxCol a SUD}
8. Quanto detto vale per il confronto con il PE a SUD per determinare il valore delle variabili minCol e maxCol; operazioni analoghe si effettuano per il confronto con il PE a NORD, a EST, a OVEST per determinare i valori delle variabili minRow e maxRow.

Di seguito si farà riferimento alla procedura, sopra descritta, con il nome di “verifica adiacenza sotto-figure con il processore a NORD”.

Quanto detto sinora vale in riferimento alla equazione (1), discorsi assolutamente analoghi vanno fatti in riferimento alle equazioni (2),(3),(4).

Bisogna vedere, a questo punto, come costruire il rettangolo circoscrivente a partire dalla composizione dei sotto-rettangoli: facendo riferimento alla Fig 4.12 e ricordando la disposizione dei PE, si vede che la figura da iscrivere è suddivisa all’interno dei PE 0, 1, 4, 5. Il lato verticale del rettangolo che circoscrive la sotto-figura indica il numero di righe facenti parte del rettangolo, mentre il lato orizzontale della sotto-figura indica il numero di colonne facenti parte del rettangolo, in riferimento al PE X si indichi con rg_x il numero di righe facenti parte del rettangolo, cioè maxRow-minRow, e con cl_x il

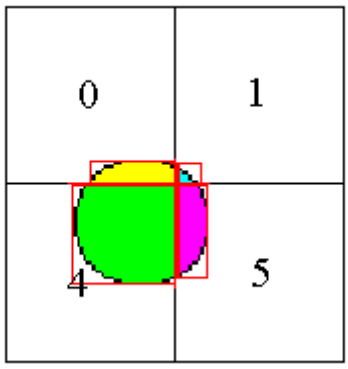


Fig 4. 12: Particolare della immagine di partenza con rettangoli che circoscrivono le sotto-regioni

numero di colonne facenti parte del rettangolo, cioè maxCol-minCol.

In riferimento al particolare esempio, l’algoritmo per determinare il rettangolo che circoscrive la figura intera è costituito dai seguenti passi:

1. Si effettua la procedura “verifica adiacenza sotto-figura con il processore a SUD”, se è positiva si valuta il max {cl_0 e cl_4} e il max {cl_1 e cl_5}.
2. Si effettua la procedura “verifica adiacenza sotto-figura con il processore a EST”, se è positiva si valuta il max {rg_0 e rg_1} e il max {rg_4 e rg_5}.

Il rettangolo che circoscrive l’intera figura ha un numero di righe e di colonne pari a:

$$RigheTOT = \max \{rg_0 \text{ e } rg_1\} + \max \{ rg_4 \text{ e } rg_5\}$$

$$ColonneTOT = \max \{cl_0 \text{ e } cl_4\} + \max \{ cl_1 \text{ e } cl_5\}$$

Quanto detto vale nel caso in cui la figura sia suddivisa soltanto tra 4 processori, se, invece, si

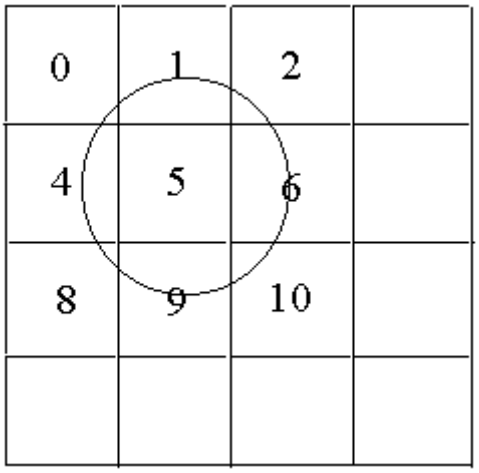


Fig 4. 13: Regione scomposta in un numero di processori superiore a quattro

verifica il caso di Fig 4.13 la regione viene suddivisa in 9 PE, nella quale si nota che nei PE 1, 5, 9 i rettangoli che circoscrivino le sotto-figure hanno il cl pari al numero totale di colonne (Nc) e nei PE 4, 5, 6 i rettangoli che circoscrivono le sotto-figure hanno l’rg pari al numero totale di righe (Nr): utilizzando questa proprietà si può adattare, con dovuti accorgimenti, la procedura “verifica adiacenza sotto-figure” anche al caso in cui il controllo di adiacenza non debba essere effettuato soltanto tra due processori: quanto detto di seguito vale in generale, qualunque sia il numero di PE occupati dalla figura da circoscrivere.

Algoritmo per la determinazione del numero di righe in ogni riga di PE della figura TOTALE:

$cont_r = 1$

loop PE_a e PE_{a+1} sono “collegati lungo l’orizzontale”?:

se sì: $cont_r$++

si valuta il max_a tra rg_a e rg_{a+1}

cl_{a+1} = Nc?

se sì: a = a + 1

vai a loop

$Righe_a = max_a\ rg_{a+i}$ $\quad i = 0,\ldots, cont_r$

con $cont_r$ si indica una variabile che tiene conto del numero di processori occupati lungo la riga di PE.

Nel caso in questione: (a=0)

0 e 1 sono collegati? Sì $cont_r$=2, calcola max_0

nel PE 1 la sotto-figura occupa tutte le colonne? Sì ⇒ ricomincia con 1 e 2

1 e 2 sono collegati? Sì $cont_r$=3

Nel PE 2 la sotto-figura occupa tutte le colonne? no ⇒ STOP

$Righe_0 = max\ (rg_0, rg_1, rg_2)$

Analogamente e parallelamente si determinano $Righe_4$ e $Righe_8$.

Algoritmo per la determinazione del numero di colonne in ogni colonna di PE della figura TOTALE:

NP_r = numero di PE lungo la riga

$cont_c$= 1

loop PE_a e PE_{a+NPr} sono “collegati lungo la verticale”?:

se sì: $cont_c$++

si valuta il max_a tra cl_a e cl_{a+NPr}

rg_{a+NPr} = Nr?

se sì: $a = a + NP_r$

vai a loop

$Colonne_a = max_a\ cl_{a+i*NPr}$ $\quad i = 0,\ldots,$ cont

con $cont_c$ si indica una variabile che tiene conto del numero di processori occupati lungo la colonna di PE.

Nel caso in questione: (a=0)

0 e 4 sono collegati? Sì $cont_c$=2, calcola max_0

nel PE 4 la sotto-figura occupa tutte le righe? Sì ⇒ ricomincia con 4 e $(4+NP_r) = (4+4) = 8$

4 e 8 sono collegati? Sì $cont_c$=3

Nel PE 8 la sotto-figura occupa tutte le righe? no ⇒ STOP

$Colonne_0 = max\ (rg_0, rg_4, rg_8)$

Analogamente e parallelamente si determinano $Colonne_1$ e $Colonne_2$

A questo punto:

Numero di righe dell'intera figura:

$RigheTOT = \Sigma\ Righe_{a+i*NPr}$ $\quad i = 0,\ldots.,\ cont_c$

Numero di colonne dell'intera figura:

$ColonneTOT = \Sigma\ Colonne_{a+i}$ $\quad i = 0,\ldots.,\ cont_r$

Alla fine del procedimento l'operazione di individuazione e di estrazione delle regioni è stata effettuata.

*4.10 Funzione **Normalizza***

Dopo aver eseguito la funzione GrowingRegioni, le figure sono tutte circoscritte; con questa funzione, in riferimento alla Fig 4.14 e a partire dalla Fig 4.15:

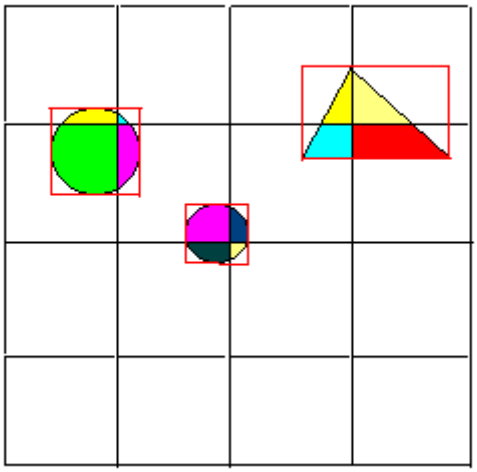


Fig 4. 14: Immagini circondate correttamente da un rettangolo circoscrivente

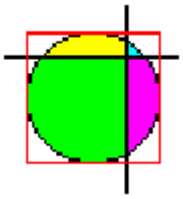


Fig 4. 15: Particolare di un'immagine circondata correttamente da un rettangolo circoscrivente

si deve costruire una figura avente la stessa forma e gli stessi colori, ma avente un numero di righe e di colonne pari rispettivamente alla somma di tutte righe e di tutte le colonne di tutti i processori: l'immagine deve, quindi, essere ingrandita di un fattore di scala da determinare a partire dal numero di colonne del rettangolo circoscrivente, e centrata.

Questa operazione deve essere effettuata serialmente rispetto all'immagine da ingrandire, considerando sempre che tutte le istruzioni vengono eseguite in parallelo su tutti i processori, anche se quelli inizialmente non contengono l'immagine da scalare.

Tenendo presente che, in ogni processore, nelle locazioni di memoria ESTREMI, ESTREMI+1, ESTREMI+2, ESTREMI+3 sono conservati i valori di, rispettivamente, minRow, maxRow, minCol, maxCol, nella funzione GrowingRegioni si pongono, come valori di inizializzazione:

minRow = NumRighe

maxRow = 0

minCol = NumColonne

maxCol = 0

all'interno di ogni processore; se almeno uno dei valori di inizializzazione di minRow, maxRow, mionCol, maxCol è stato modificato, vuol dire che il processore contiene almeno una sotto-regione da analizzare.

Si devono ora determinare il numero di righe e di colonne totali: all'interno di tutti i processori che hanno almeno una regione, per ogni sotto-regione che contengono, si devono determinare il numero di righe e di colonne del rettangolo che circoscrive l'intera figura.

Si supponga di utilizzare con una matrice di processori 4x4 e di avere una situazione del tipo di Fig 4.16:

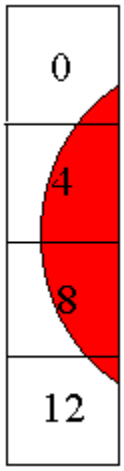


Fig 4. 16: Parte a sinistra di un'immagine che occupa quattro righe di PE

Si indichi con NR_x, il numero di righe della figura all'interno del PE X e si supponga che:

$NR_0 = 4$, $NR_4 = 10$, $NR_8 = 10$ e $NR_{12} = 3$.

Per determinare il numero delle righe, si trasferisca il numero di righe presenti in ogni PE ai PE a SUD.

Si utilizzino le seguenti variabili:

Proprio = il numero di righe della sotto-regione contenuta nel processore; variabile già nota all'interno di ogni PE.

Passato = il numero di righe della sotto-regione contenuta nel processore del PE a SUD ottenuto dal PE a SUD.

DaPassare = il numero di righe della sotto-regione contenuta nel processore proprio da passare al PE a SUD.

Totale = il numero di righe della sotto-regione contenuta nel processore Totale all'iterata totale.

Il ciclo da eseguire è il seguente:

Passato = 0

Totale = Proprio

DaPassare = Proprio

LOOP

$Passato_{PE} = DaPassare_{PE\_SUD}$

$Totale_{PE} = Totale_{PE} + Passato_{PE}$

$DaPassare_{PE} = Passato_{PE}$

Il PE è a SUD?

Sì: SLEEP

maxRow ≠ NumRighe-1(è il caso del PE 12)?

Sì: SLEEP

Se almeno un PE è attivo vai a LOOP

Ad ogni iterata, si pone nello stato di SLEEP un processore se si verifica almeno una delle due condizioni:

1. la sotto-regione non occupa tutte le righe (maxRow ≠ NumRighe –1)
2. tra tutti i PE che contengono una sotto-regione della figura di partenza, il PE più a SUD è nello stato di SLEEP.

Le iterate relative all'esempio, vengono di seguito analizzate e la riga segnata in rosso è quella relativa al PE che nella particolare iterata viene posto nello stato di SLEEP:

Prima ITERATA:

PE_0:	Proprio = 4	Passato = 0	DaPassare = 4	Totale = 4
PE_4:	Proprio = 10	Passato = 0	DaPassare = 10	Totale = 10
PE_8:	Proprio = 10	Passato = 0	DaPassare = 10	Totale = 10
PE_{12}:	Proprio = 3	Passato = 0	DaPassare = 3	Totale = 3

Seconda ITERATA:

PE_0:	Proprio = 4	Passato = 10	DaPassare = 10	Totale = 14
PE_4:	Proprio = 10	Passato = 10	DaPassare = 10	Totale = 20
PE_8:	Proprio = 10	Passato = 3	DaPassare = 3	Totale = 13

Terza ITERATA:

PE_0:	Proprio = 4	Passato = 10	DaPassare = 10	Totale = 24
PE_4:	Proprio = 10	Passato = 3	DaPassare = 3	Totale = 23

Quarta ITERATA:

PE_0:	Proprio = 4	Passato = 3	DaPassare = 3	Totale = 27

COMPLESSIVAMENTE:

PE_1:	Proprio = 4	Passato = 3	DaPassare = 3 Totale = 27
PE_4:	Proprio = 10	Passato = 3	DaPassare = 3 Totale = 23
PE_8:	Proprio = 10	Passato = 3	DaPassare = 3 Totale = 13
PE_{12}:	Proprio = 3	Passato = 0	DaPassare = 3 Totale = 10

E' necessario, ora, effettuare il procedimento duale al precedente avente lo scopo di trasferire il numero di righe presenti in ogni PE ai PE a **NORD**. Si pongano:

Proprio = il numero di righe della sotto-regione contenuta nel processore; variabile già nota all'interno di ogni PE.

Passato = il numero di righe della sotto-regione contenuta nel processore del PE a SUD ottenuto dal PE a NORD.

DaPassare = il numero di righe della sotto-regione contenuta nel processore proprio da passare al PE a NORD.

Totale = il numero di righe della sotto-regione contenuta nel processore Totale all'iterata totale.

Il ciclo da eseguire è il seguente:

Passato = 0

Totale = Proprio

DaPassare = Proprio

LOOP

$Passato_{PE} = DaPassare_{PE\_SUD}$

$Totale_{PE} = Totale_{PE} + Passato_{PE}$

$DaPassare_{PE} = Passato_{PE}$

Il PE è a NORD?

Sì: SLEEP

minRow ≠ 0(è il caso del PE 0)?

Sì: SLEEP

Se almeno un PE è attivo vai a LOOP

Questo loop si effettua subito dopo il loop precedente, quindi i valori iniziali delle variabili sono quelli finali del loop precedente e, analogamente a prima, si ottiene:

Prima ITERATA:

PE_0:	Proprio = 4	Passato = 0	DaPassare = 4	Totale = 27
PE_4:	Proprio = 10	Passato = 0	DaPassare = 10	Totale = 23
PE_8:	Proprio = 10	Passato = 0	DaPassare = 10	Totale = 13
PE_{12}:	Proprio = 3	Passato = 0	DaPassare = 3	Totale = 3

Seconda ITERATA:

PE_4:	Proprio = 10	Passato = 4	DaPassare = 4	Totale = 27
PE_8:	Proprio = 10	Passato = 10	DaPassare = 10	Totale = 23
PE_{12}:	Proprio = 3	Passato = 10	DaPassare = 10	Totale = 13

Terza ITERATA:

PE_8:	Proprio = 10	Passato = 4	DaPassare = 4	Totale = 27
PE_{12}:	Proprio = 3	Passato = 10	DaPassare = 10	Totale = 23

Quarta ITERATA:

PE_{12}: Proprio = 3 Passato = 4 DaPassare = 10 Totale = 27

COMPLESSIVAMENTE:

PE_0: Proprio = 4 Passato = 0 DaPassare = 4 Totale = 27

PE_4: Proprio = 10 Passato = 4 DaPassare = 4 Totale = 27

PE_8: Proprio = 10 Passato = 4 DaPassare = 4 Totale = 27

PE_{12}: Proprio = 3 Passato = 4 DaPassare = 10 Totale = 27

Come si vede, alla fine dei due procedimenti, tutti i processori lungo la stessa colonna di PE contengono al loro interno il numero totale di righe della figura.

Quanto detto viene effettuato in parallelo su tutte le colonne le quali daranno tutte lo stesso risultato perché il conteggio viene fatto in relazione alle colonne presenti nel rettangolo che circoscrive l'intera figura e non sulla semplice figura.

In relazione alla Fig 4.17, per quanto detto prima, in parallelo nei PE 0, 4, 8, 12 e nei PE 1, 5, 9, 13, vengono effettuati i due loop di cui sopra e lo stesso vale per le altre due colonne; complessivamente, in tutti i 16 PE viene memorizzato il numero totale di righe nella figura che, per quanto precedentemente detto, è lo stesso.

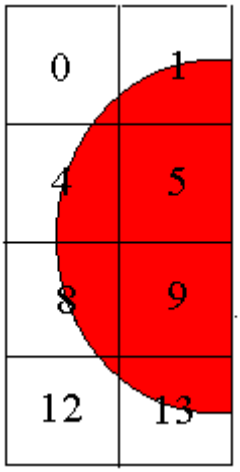


Fig 4. 17: Rappresentazione della prima e della seconda colonna di PE occupate dalla regione di partenza

Quanto finora detto, vale per il conteggio delle RIGHE nel rettangolo circoscrivente, procedimento assolutamente analogo si effettua per il conteggio delle COLONNE.

Alla fine i 16 PE conterranno al loro interno due variabili per ogni sotto-regione che contengono:

1. Numero TOTALE Righe nella figura = $N\_TOT\_FIG_R$
2. Numero TOTALE Colonne nella figura = $N\_TOT\_FIG_C$

A questo punto, dalla pila ESTREMI si eliminano i quattro estremi considerati, visto che nel considerare i PE a N-E-W-S sono stati tutti considerati in parallelo; per eliminarli basta semplicemente porre i loro valori uguali a quelli di inizializzazione.

Si faccia, ora, riferimento alla Fig 4.18:

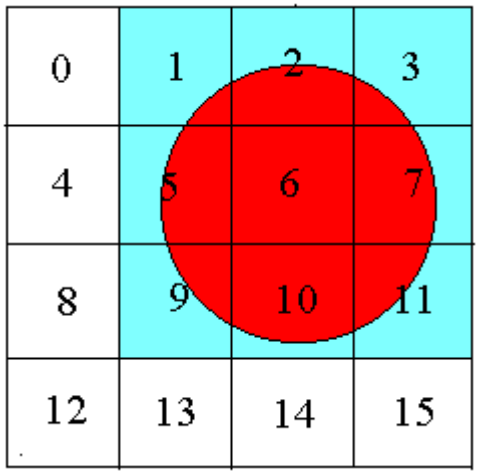


Fig 4. 18: Processore occupati dall'immagine di partenza

Supponendo che la figura da circoscrivere sia quella in rosso, tutti i PE interessati sono quelli colorati in azzurro; ognuno di questi PE hanno al loro interno il numero di righe della figura ($N\_TOT\_FIG_R$). Quanto sarà detto riguardo le RIGHE vale analogamente per le COLONNE.

Se si indica con:

1. Numero TOTALE Righe in tutti i PE = N_TOT_R
2. Numero TOTALE Colonne in tutti i PE = N_TOT_C

Si deve effettuare una scalatura lungo le righe e lungo le colonne, con i seguenti fattori di scala:

1. $N\_TOT_R : N\_TOT\_FIG_R$
2. $N\_TOT_C : N\_TOT\_FIG_C$

In realtà, siccome le figure da individuare (cerchio e triangolo equilatero) sono simmetriche lungo l'orizzontale e lungo la verticale, il rettangolo circoscrivente di cui si è parlato deve essere un quadrato, quindi prima di avviare tutta la procedura seguente è necessario fare il controllo che $N\_TOT\_FIG_R = N\_TOT\_FIG_c$, con una certa tolleranza, se così non fosse non si prosegue. I due fattori di scala devono quindi essere uguali perché il numero di righe e di colonne nella figura devono essere uguali e, poiché il numero di PE lungo l'orizzontale e lungo la verticale devono essere uguali e le righe e le colonne all'interno di un PE devono essere uguali, anche il numero di righe totali a disposizione deve essere uguale al numero di colonne.

A questo punto, all'interno di ogni processore contenente una sotto-figura sono contenuti i valori del numero di righe e di colonne costituenti l'intera figura: da una situazione del tipo di Fig 4.19:

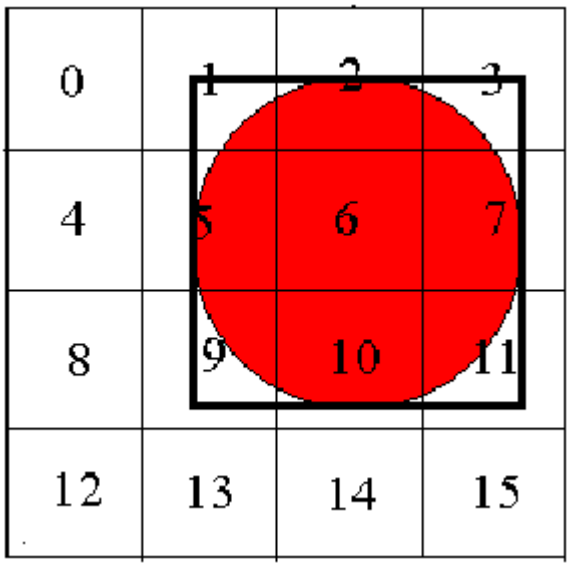


Fig 4. 19: Immagine di partenza e relativo rettangolo circoscrivente

si deve passare ad una situazione del tipo di Fig 4.20.

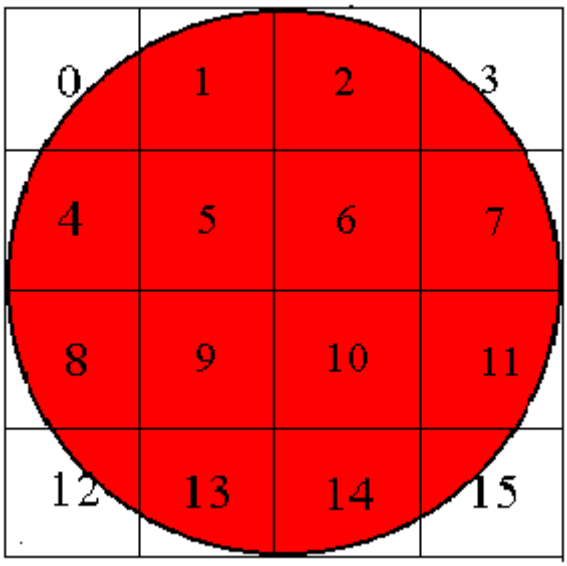


Fig 4. 20: Immagine finale

Con la ripetizione dell'istruzione XFER, si può trasferire l'immagine in alto a six all'interno della matrice di processori: si effettua un trasferimento lungo la riga di PE in modo tale che la riga verticale a six (nei PE 1, 5, 9) venga trasferita nei processori (0, 4, 8) mantenendo lo stesso valore di

minCol e automaticamente questo accade per l'intera figura; si deve mantenere la corrispondenza dell'indice dei pixel nel passaggio da un PE all'altro con l'istruzione XFER.

Analogo discorso per le colonne.

Si otterrà una situazione del tipo di Fig 4.21:

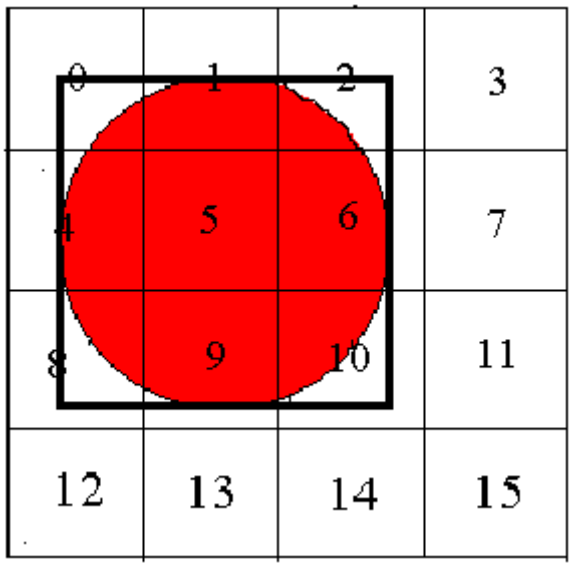


Fig 4. 21: Immagine ottenuta dopo la traslazione lungo i PE

La figura è tutta traslata nei PE in alto a six.

Se il lato sinistro del rettangolo, stava nel PE X_R e il PE più a six nella stessa riga di PE è PE Y_R, allora tutti i pixel della figura devono essere traslati nei PE a six di X_R-Y_R; nell'esempio precedente:

PE X_R=1 e PE Y_R=0 → 1-0=1;

PE X_R=5 e PE Y_R=4 → 5-4=1;

PE X_R=9 e PE Y_R=8 → 9-8=1;

analogo discorso per la traslazione lungo le colonne.

Fatto ciò, la figura è nei PE più in alto e a six; si deve fare in modo, ancora, di ottenere una situazione come quella della Fig 4.22 in cui il lato sinistro della figura coincida con il lato sinistro della "matrice di PE" e il lato in alto della figura coincida con il lato in alto della "matrice di PE". Si faccia riferimento al caso del lato sinistro, il caso del lato in alto è analogo. Se nella figura precedente il lato sinistro è individuato dalla variabile minCol, per ottenere una situazione come quella della figura sottostante, è necessario effettuare un numero di traslazioni a six pari a minCol,

con la solita attenzione al fatto che nell'immagine l'indice è bidimensionale, uno per le righe ed uno per le colonne, mentre all'interno del PE l'indice che individua il numero del pixel è lineare.

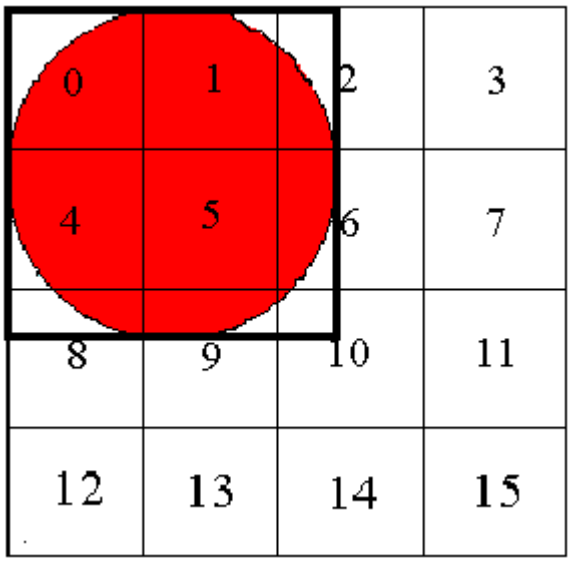


Fig 4. 22: Immagine ottenuta dopo la traslazione totale

Dato che la traslazione riguarda l'intera figura, la traslazione deve essere effettuata distinguendo il caso in cui il pixel trasli in un processore "interno" al PE e quello in cui il pixel trasli in un PE ad EST, per far ciò si veda il paragrafo 4.2.

Traslata completamente la figura in alto a six, si deve effettuare la scalatura: tenendo presente che l'immagine totale è suddivisa all'interno dei processori e dopo la scalatura sarà contenuta ancora in processori diversi, effettuare la scalatura con un solo fattore di moltiplicazione complica notevolmente il trasferimento dell'immagine all'interno dei processori; d'altra parte, l'utilizzo di fattori di scala diversi causa una deformazione della figura: si sceglie di utilizzare fattori di moltiplicazioni diversi per pixel appartenenti a processori diversi, ma in modo tale da ottenere un buon compromesso tra numero necessario di trasferimenti di pixel e la deformazione conseguente all'utilizzo di fattori di scala diversi; infatti, considerato che le immagini da individuare sono molto diverse da tutte le altre, una lieve deformazione dell'immagine non può creare ambiguità. Invece della Fig 4.20, si ottiene qualcosa del tipo di Fig 4.23, in cui le possibili deformazioni sono, però, più accentuate per rendere più chiara il tipo di problema.

L'algoritmo che si è scelto è il seguente:

Algoritmo di scalatura approssimata:

- Il numero totale di righe nella matrice di PE (N_{TOT_R}) è pari al numero di righe all'interno di un PE (N_{R_PE}) per il numero totale di righe di PE ($N_{PE\_TOT\_R}$):

$$N_{TOT\_R} = N_{R\_PE} * N_{PE\_TOT\_R}$$

- Ogni PE contiene il numero totale di righe della figura $N_{TOT\_FIG\_R}$

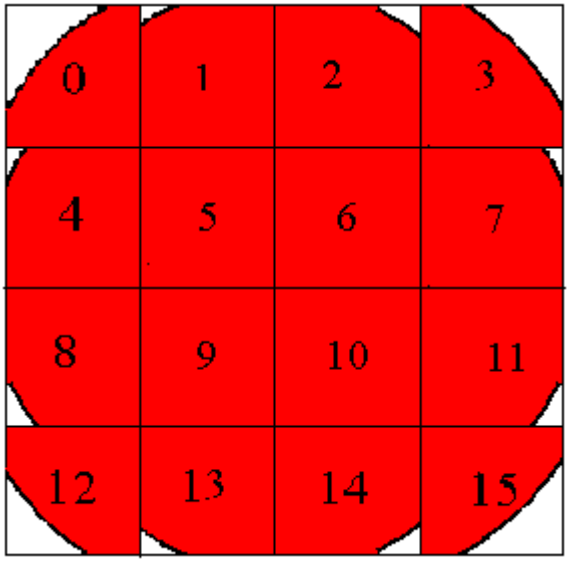


Fig 4. 23: Immagine ottenuta dopo la scalatura approssimata

- La figura occupa un numero di righe di PE ($N_{PE\_FIG\_R}$) pari a:

$$N_{PE\_FIG\_R} = \lceil N_{TOT\_FIG\_R} : N_{R\_PE} \rceil$$

- L'ultimo PE contiene un numero di righe ($N_{ULTIMO\_PE\_R}$) pari al valore di maxRow di tale PE, pari a:

$$N_{ULTIMO\_PE\_R} = N_{TOT\_FIG\_R} - (N_{PE\_FIG\_R} - 1) * N_{R\_PE}$$

- Per prima cosa si scalano i valori relativi all'ultimo PE, determinando il fattore di moltiplicazione (FM):

$$FM = \lfloor N_{TOT\_R} : N_{TOT\_FIG\_R} \rfloor$$

- Il numero di righe che deve occupare l'ultimo PE (RO_{ULT_PE}) è

$$RO_{ULT\_PE} = N_{ULTIMO\_PE\_R} * FM$$

- Si determina il numero di PE occupati dalla espansione delle righe relative all'ultimo PE ($PE_{OCC\_ULT\_PE\_R}$):

$$PE_{OCC\_ULT\_PE\_R} = \lceil RO_{ULT\_PE} : N_{R\_PE} \rceil$$

- Per valutare la parte frazionaria di PE occupati (PF_R):

$$PF_R = RO_{ULT\_PE} \,\%\, N_{R\_PE}$$

- Se $PF_R < (N_{R\_PE} :2)$

$$PE_{OCC\_ULT\_PE} = PE_{OCC\_ULT\_PE\_R} - 1$$

- I PE rimasti (PE_{RM_R}) sono:

$$PE_{RM\_R} = PE_{iniziali\_R} - PE_{OCC\_ULT\_PE\_R}$$

- La figura occupa un numero di righe di PE ($PE_{RM\_FIG\_R}$):

$$PE_{RM\_FIG\_R} = N_{PE\_FIG\_R} - PE_{OCC\_ULT\_PE\_R}$$

di cui :

a. i primi $(PE_{RM\_R} \,\%\, PE_{RM\_FIG\_R})$ PE $\Rightarrow$ occupano $(PE_{RM\_R} / PE_{RM\_FIG\_R} +1)$ PE.

b. i secondi $(PE_{RM\_R} - (PE_{RM\_R} \,\%\, PE_{RM\_FIG\_R}))$ PE $\Rightarrow$ occupano $(PE_{RM\_R} / PE_{RM\_FIG\_R})$ PE.

c. infine ci sono i PE relativi all'ultimo PE di partenza.

ESEMPIO 1

Si supponga di avere una regione circoscritta dal rettangolo di Fig 4.24:

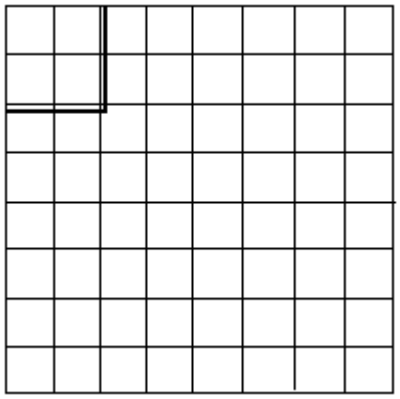


Fig 4. 24: Rettangolo circoscrivente

in cui la matrice di processori è 8x8 ed in ogni PE ci sono 16 pixel (quindi la matrice in terna al processore è 4x4). Il rettangolo circoscrivente contiene 9 righe, quindi:

- $N_{R\_PE} = 4$
- $N_{PE\_TOT\_R} = 8$
- $N_{TOT\_R} = 4 * 8 = 32$
- $N_{TOT\_FIG\_R} = 9$
- $N_{PE\_FIG\_R} = \lceil 9 : 4 \rceil = \lceil 2.25 \rceil = 3$
- $N_{ULTIMO\_PE\_R} = 9 – (3 -1) * 4 = 1$
- $FM = \lfloor 32 : 9 \rfloor = \lfloor 3.55 \rfloor = 3$
- $RO_{ULT\_PE} = 1 * 3 = 3$
- $PE_{OCC\_ULT\_PE\_R} = \lceil 3 : 4 \rceil = \lceil 0.75 \rceil = 1$
- $PF_R = 3 \% 4 = 3$
- $3 < (4 :2) = 2$? NO
- $PE_{RM\_R} = 8 – 1 = 7$
- $PE_{RM\_FIG\_R} = 3 – 1 = 2$
- $7 \% 2 = 1$

RIEPILOGANDO:

- i PE nelle prime (7%2)=1 righe occupano (7/2+1)=4 PE
- i PE nelle seconde (2- (7%2))=1 righe occupano 7/2=3 PE
- 1 riga del PE nella terza riga di PE occupa 3 righe di 1 PE

In totale (4 + 3 + 1) PE= 8 PE:

- il PE della prima riga conteneva 4 righe, ora occupa 4 * 4 = 16 righe (FM = 4)
- il PE della seconda riga conteneva 4 righe, ora occupa 4 * 3 = 12 righe (FM = 3)
- il PE della terza riga conteneva 1 riga, ora occupa1 * 3 = 3 righe (FM = 3)

ESEMPIO 2

Si supponga di avere una situazione in cui la matrice di processori è 16x16 ed in ogni PE ci sono 64 pixel (quindi la matrice in terna al processore è 8x8). Il rettangolo circoscrivente contiene 53 righe, quindi:

- $N_{R\_PE} = 8$
- $N_{PE\_TOT\_R} = 16$
- $N_{TOT\_R} = 16 * 8 = 128$
- $N_{TOT\_FIG\_R} = 53$
- $N_{PE\_FIG\_R} = \lceil 53 : 8 \rceil = \lceil 6.62 \rceil = 7$
- $N_{ULTIMO\_PE\_R} = 53 – (7 -1) * 8 = 5$
- $FM = \lfloor 128 : 53 \rfloor = \lfloor 2.4 \rfloor = 2$
- $RO_{ULT\_PE} = 2 * 5 = 10$
- $PE_{OCC\_ULT\_PE\_R} = \lceil 10 : 8 \rceil = \lceil 1.25 \rceil = 2$
- $PF_R = 10 \% 8 = 2$
- $2 < (8 : 2) = 4$? Sì $\Rightarrow PE_{OCC\_ULT\_PE\_R} = 2 - 1 = 1$
- $PE_{RM\_R} = 16 – 1 = 15$
- $PE_{RM\_FIG\_R} = 7 – 1 = 6$
- $15 \% 6 = 3$

RIEPILOGANDO:

- i PE nelle prime $(15 \% 6) = 3$ righe di PE $\Rightarrow$ occupano $(15 / 6 + 1) =$ 3 PE.
- i PE nelle seconde $(6 - (15 \% 6)) = 3$ righe di PE $\Rightarrow$ occupano $(15 / 6) = 2$ PE.
- infine ci sono i PE relativi all’ultimo PE di partenza.

In totale (3*3+3*2+1)= 16 PE:

- ogni PE della prima, seconda, terza riga di PE nella matrice di PE conteneva 8 righe, ora ne occupa 3 * 8 = 24 righe (FM = 3).
- ogni PE della quarta, quinta, sesta riga di PE nella matrice di PE conteneva 8 righe, ora ne occupa 2 * 8 = 16 righe (FM = 2).
- il PE della settima riga di PE nella matrice di PE: conteneva 5 righe, ora ne occupa 8 righe (FM = 8/5=1.6).

Come si vede la differenza tra i FM e la conseguente deformazione è minima.

Fatto ciò, è noto in quali PE verrà mappata la sotto-immagine relativa ad ogni PE di partenza, si devono, ora, distribuire le righe di ogni PE in ognuno dei PE corrispondenti. Nel caso dell'ESEMPIO 1:

1. il PE nella terza riga deve occupare 1 PE, occupa l'ottavo PE (passa i valori 1 e 8 al PE sovrastante). Si copia il contenuto del PE nella riga 3 nel PE nella riga 8.
2. il PE nella seconda riga deve occupare 3 PE, occupa dal PE nella riga (8-1) = 7 al PE nella riga (7 –3 +1) = 5 (passa i valori 3 e 7 al PE sovrastante). Si copia il contenuto del PE nella riga 2 nel PE nella riga 5.
3. il PE nella prima riga deve occupare 4 PE, occupa dal PE nella riga (7-3) = 4 al PE nella riga (4 –4 +1) = 1. Si copia il contenuto del PE nella riga 1 nel PE nella riga 1.

- PE della terza (3) riga: dalla terza (3) all'ottava (8) riga ⇒ trasferisce di 8 – 3 = 5 righe di PE.
- PE della seconda (2) riga: dalla seconda (2) alla quinta-settima (5-7) ⇒ trasferisce di 5 – 2 = 3 righe di PE e poi copia nei 7-5 = 2 successivi.
- PE della prima (1) riga: dalla prima (1) alla prima-quarta (1-4) ⇒ trasferisce di 1 – 1 = 0 righe di PE e poi copia nei 4-1 = 3 successivi.

Sempre in riferimento all'ESEMPIO 1, si supponga di volere analizzare il mapping tra processori dovuto alla scalatura, si consideri il PE nella seconda riga, come in Fig 4.25.

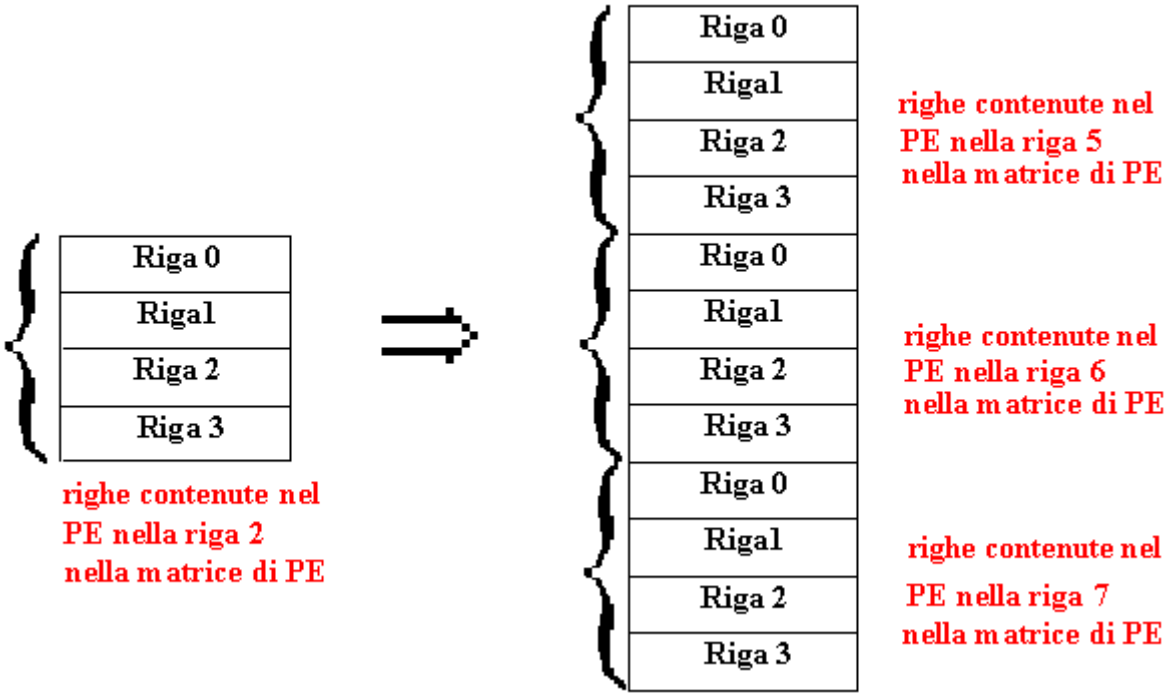


Fig 4. 25: Mapping delle righe del processore nella seconda riga di PE

Il modo in cui le 4 righe del PE:

- La riga 0 viene copiata, previa traslazione in basso, nella riga (0*3)%4 = 0%4= 0 del PE (0*3/4)+4= (0/4)+4=4: Riga 0 → Riga 0 del PE 4.
- La riga 1 viene copiata, previa traslazione in basso, nella riga (1*3)%4 = 3%4= 3 del PE (1*3/4)+4=(3/4)+4=4: Riga 1 → Riga 3 del PE 4.
- La riga 2 viene copiata, previa traslazione in basso, nella riga (2*3)%4 = 6%4= 2 del PE (2*3/4)+4= (6/4)+4=5: Riga 2 → Riga 2 del PE 5
- La riga 3 viene copiata, previa traslazione in basso, nella riga (3*3)%4 = 9%4= 1 del PE (3*3/4)+4= (9/4)+4=6: Riga 3 → Riga 1 del PE 6

Il numero 3 si utilizza perché il numero di processori associati al PE della riga 2 è proprio 3.

Si noti anche che si deve ogni volta fare la distinzione tra il caso in cui la riga sottostante faccia parte dello stesso PE e quello in cui faccia parte del PE sottostante.

E' necessario,quindi, un loop esterno (**loop_ex**) avente un numero di iterate pari al numero di righe contenute in ogni PE, il quale, all'interno di ogni iterata, effettua un numero di shift verso il basso pari al fattore di moltiplicazione relativo al singolo PE: è possibile ottenere questo risultato con un

loop interno (**loop_in**) avente un numero di iterate pari al fattore di moltiplicazione relativo al singolo PE, all'interno del quale si effettua uno shift unitario verso il basso, poiché questo è un numero diverso all'interno dei vari PE, il numero di iterate del loop_in è pari al *massimo* numero di PE associati ad ognuno dei PE di partenza-1.

Per come è stato costruito l'algoritmo, il PE più in alto (PE_0) ha sempre il massimo numero di PE associati, questo valore viene posto in un registro del controllore e funge da contatore per il loop_in.

Sempre in riferimento all'esempio 1, si osservi la Fig 4.26.

I vari passi da effettuare sono:

1. Del PE 0 si copia:
    a. la riga 0 nella riga 0*4=0, cioè nella riga 0%4=0 del PE 0/4=0
    b. la riga 1 nella riga 1*4=4, cioè nella riga 4%4=0 del PE 4/4=1
    c. la riga 2 nella riga 2*4=8, cioè nella riga 8%4=2 del PE 8/4=2
    d. la riga 3 nella riga 3*4=12, cioè nella riga 12%4=3 del PE 12/4=3

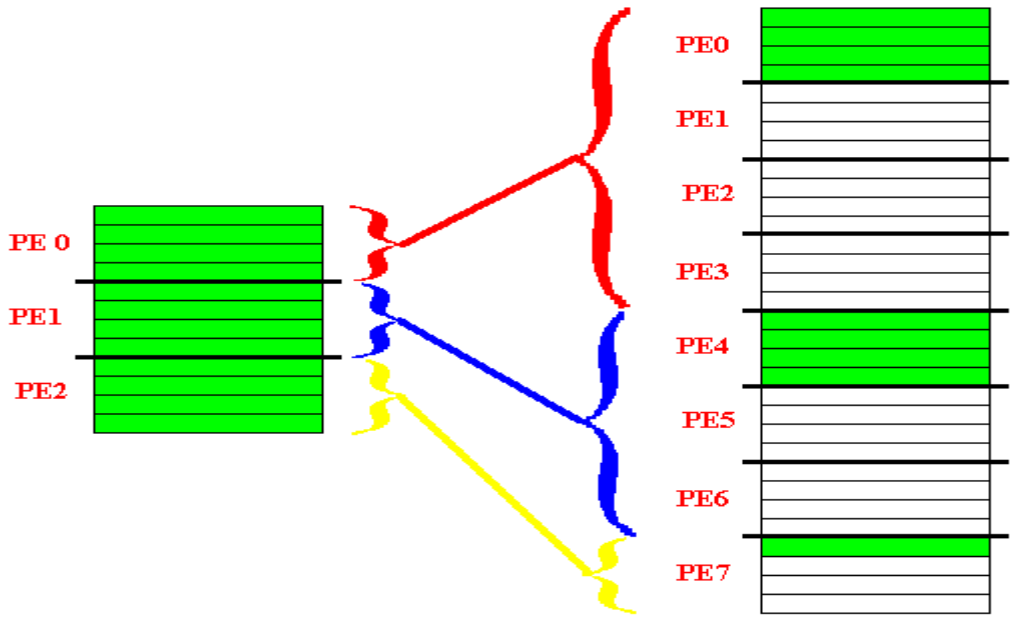


Fig 4. 26: Mapping dai PE contenenti la figura di partenza nei PE di destinazione

2. Del PE 1 si copia:
    a. la riga 0 nella riga 0*3=0, cioè nella riga 0%4=0 del PE 0/4+4=4

b. la riga 1 nella riga 1*3=3, cioè nella riga 3%4=3 del PE 0/4+4=4

c. la riga 2 nella riga 2*3=6, cioè nella riga 6%4=2 del PE 6/4+4=5

d. la riga 3 nella riga 3*3=9, cioè nella riga 9%4=1 del PE 9/4+4=6

3. Del PE 2 si copia:

   a. la Riga 0 nella riga 0*3=0, cioè nella riga 0%4=0 del PE 0/4+7=7

Nel seguito vengono riportati i numeri di traslazioni da effettuare:

PE 0:	0→0	0-0=0
	1→4	4-1=3
	2→8	8-2=6
	3→12	12-3=9
PE 1:	0→0	0-0=0
	1→3	3-1=2
	2→6	6-2=4
	3→9	9-3=6
PE 2:	0→0	0-0=0

Il massimo valore di traslazione è pari a 9 e si ottiene dal massimo valore di PE associati (è sempre uguale al numero di PE associati al primo PE moltiplicato per il numero di righe in ogni PE -1), cui si sottrae il numero di righe in ogni PE-1, nel caso specifico, infatti, si ha:

$$4+(4-1)-(4-1)=4*3-3=9$$

Il loop_ex ha un numero di iterate pari al numero di righe contenute in ogni PE, nel caso dell'ESEMPIO 1, quindi, sono 4:

ITERATA 1:

Shift di:

4*0-0=0 per il PE 0

3*0-0=0 per il PE 1

4*0-0=0 per il PE 2

ITERATA 2:

Shift di:

4*1-1=3 per il PE 0

3*1-1=2 per il PE 1

(in realtà vengono effettuate 2 iterate del loop_in per la traslazione di due, si pone il PE1 nello stato di SLEEP e si effettuano altri 4-3=1 shift).

In totale si hanno 4*1-1=3 iterate di loop_in, si ottiene la situazione di Fig 4.27, in cui gli shift si effettuano a partire dalle righe colorate in verde e contenenti il numero rosso corrispondente alla riga da traslare.

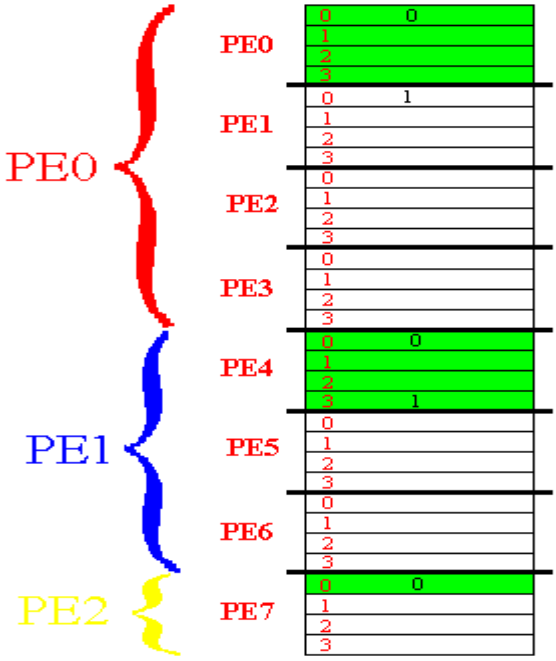


Fig 4. 27: Righe contenute nei PE dopo la seconda iterata del loop_ex

ITERATA 3:

Shift di:

4*2-2=6 per il PE 0

3*2-2=4 per il PE 1

(in realtà vengono effettuate 4 iterate del loop_in per la traslazione di quattro, si pone il PE1 nello stato di SLEEP e si effettuano un altri 6-4=2 shift).

In totale si hanno 4*2-2=6 iterate di loop_in, si ottiene la situazione della seconda Fig 4.28, in cui gli shift si effettuano a partire dalle righe colorate in verde e contenenti il numero rosso corrispondente alla riga da traslare.

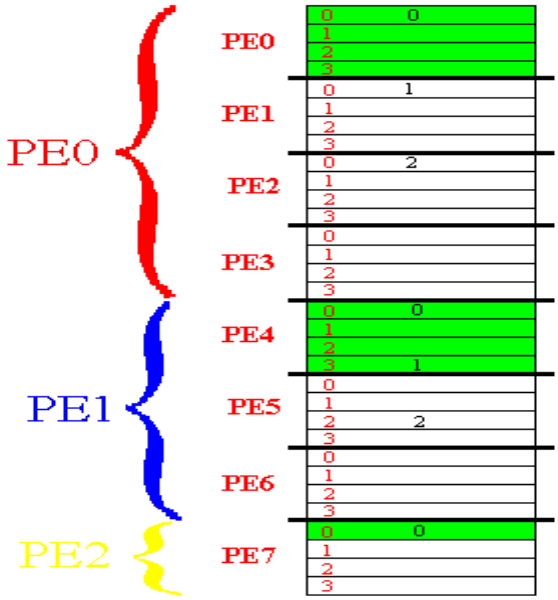


Fig 4. 28: Righe contenute nei PE dopo la terza iterata del loop_ex

ITERATA 4:

Shift di:

4*3-3=9 per il PE 0

3*3-3=6 per il PE 1

(in realtà vengono effettuate 6 iterate del loop_in per la traslazione di sei, si pone il PE1 nello stato di SLEEP e si effettuano un altri 9-6=3 shift).

In totale si hanno 4*3-3=9 iterate di loop_in, si ottiene la situazione di Fig 4.29, in cui gli shift si effettuano a partire dalle righe colorate in verde e contenenti il numero rosso corrispondente alla riga da traslare.

Facendo riferimento alla Fig 4.30, si deve copiare il contenuto delle righe colorate in giallo, nei PE a SUD effettuando un numero di iterate pari al numero di PE associati al PE di partenza –1; il numero di iterate è pari al numero di PE associati al PE0, ponendo di volta in volta i PE nello stato di SLEEP associati ai PE di partenza con un numero di PE associati minore, facendo un discorso a parte per l'ultimo PE, cui è associato un numero di righe "frazionario". Il numero di PE associati al PE 0, nell'esempio, è 4, quindi si devono effettuare 4-1=3 iterate, riportate nella Fig 4.31, 4.32, 4.33 in cui i numeri interni in *grigio* rappresentano i numeri delle righe che vengono inseriti ad ogni iterata.

La configurazione finale è quello di Fig 4.33.

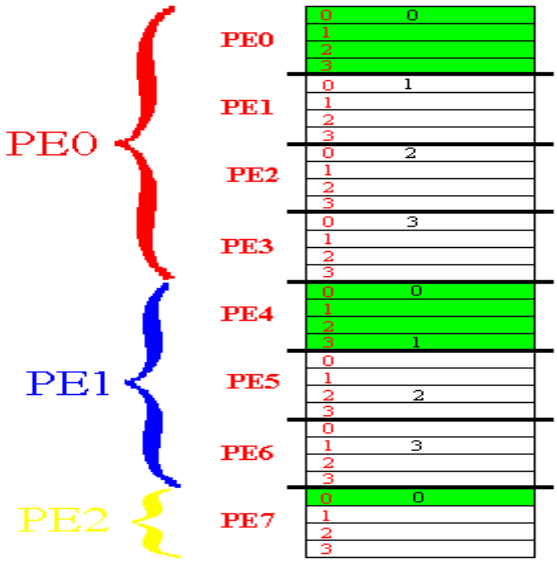


Fig 4. 29: Righe contenute nei PE dopo la quarta iterata del loop_ex

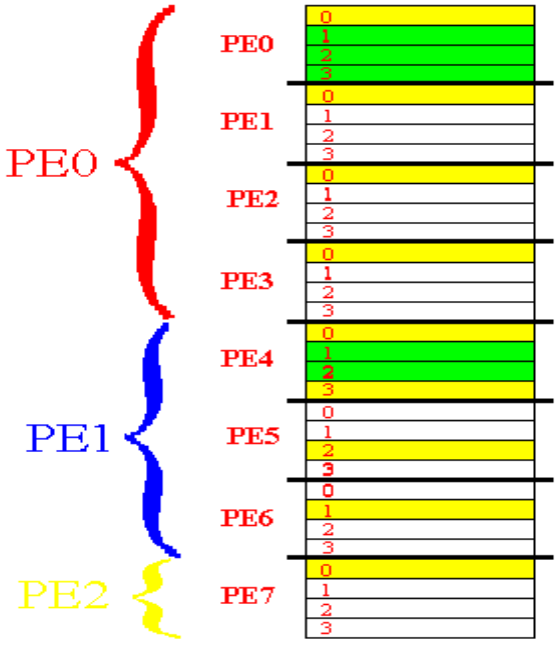


Fig 4. 30: Situazione iniziale per la fase di copia nei PE a SUD

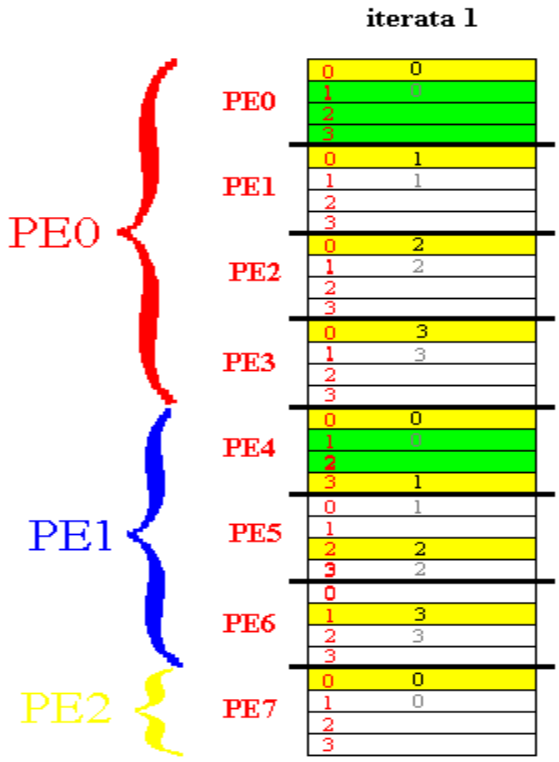


Fig 4. 31: Situazione dopo la prima iterata del ciclo relativo alla fase di copia nei PE a SUD

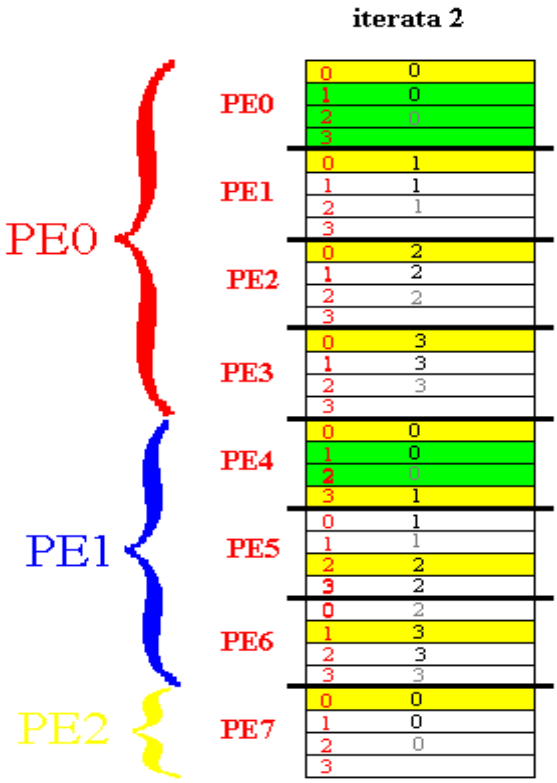


Fig 4. 32: Situazione dopo la seconda iterata del ciclo relativo alla fase di copia nei PE a SUD

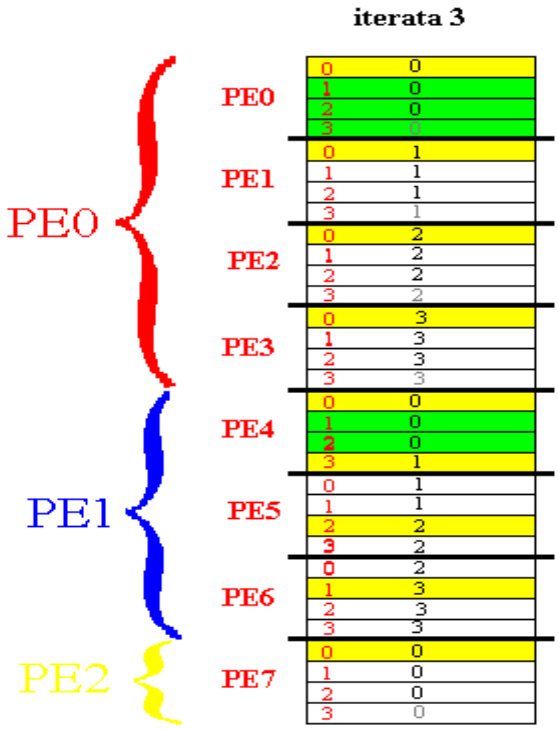


Fig 4. 33: Situazione dopo la terza iterata del ciclo relativo alla fase di copia nei PE a SUD

*4.11 Funzione **Correla***

La funzione serve a determinare il grado di similitudine tra l'immagine appena scalata ed ognuno dei modelli rappresentanti i segnali stradali, al fine di determinare il più simile.
Si suppone di utilizzare un insieme indicizzato di immagini ognuna dei quali è un nodello di segnali stradali, si deve rilevare l'immagine e memorizzare l'indice dell'immagine più simile all'immagine scalata: quanto detto è possibile utilizzando l'algoritmo di *Vector Quantization* già implementato e testato [12], avente proprio lo scopo di individuare l'immagine più simile, all'interno di un gruppo di immagini, ad un'immagine data.

5. Un esempio di implementazione

5.1 Introduzione

Questa sezione riguarda l'implementazione dell'algoritmo di conversione tra i due sistemi di rappresentazione del colore RGB→HSV su simulatore SIMPil.

Il simulatore riceve in ingresso due file:

- *.spa: contiene il codice Assembler.
- *.blf: è un file di testo in cui ogni riga rappresenta il nome di un'immagine (*.bmp) in input: ad ogni chiamata dell'istruzione SAMPLE viene campionata l'immagine relativa alla riga corrente nel file *.blf.

Come è noto SIMPil è un'architettura SIMD, è quindi costituita da una rete di calcolatori a MESH.

Si suppone di dividere l'immagine di partenza in sotto-immagini e far elaborare la singola porzione di immagine ad un processore dell'array, secondo lo schema di Fig 3.1.

Supponendo di avere un'immagine costituita da 320x240 pixel, si associa ad ogni PE una porzione di immagine, di default 4x4 per un totale di 16 pixel: questo è il Pixel per Processor Element (PPE):

sono necessari 320/4=80 processori lungo la direzione orizzontale e 240/4=60 processori lungo la direzione verticale, ognuno dei quali esegue le istruzione sulla propria parte di dati in parallelo: per elaborare l'immagine di partenza, non è necessario il tempo per l'elaborazione di 320x240 pixel, come nel caso seriale, ma un tempo pari all'elaborazione di 4x4 pixel cui si aggiungono i tempi per la comunicazione e per il controllo da parte dell'unità centrale: i tempi di calcolo sono notevolmente ridotti.

Il PPE è un parametro fondamentale perché definisce la relazione tra la granularità dei PE e la quantità dei dati che deve elaborare: nel seguito verrà eseguito un esame comparativo delle prestazioni ottenute con l'algoritmo implementato al variare del PPE, considerato che, al variare del PPE, anche il numero di processori nella direzione orizzontale e verticale devono essere modificati.

A tal fine si tenga presente che:

- 2 ≤ X_SIZE: numero di processori in direzione orizzontale (16 per default)
- 2 ≤ Y_SIZE: numero di processori in direzione verticale (16 per default)
- 1 ≤ PPE ≤ 32K (il PPE deve essere una potenza pari di 2)

Viene riportata una tabella contenente le relazioni tra PPE, X_SIZE, Y_SIZE per analizzare un'immagine 320x240 pixel:

PPE	X_SIZE	Y_SIZE
1	320	240
4	160	120
16	80	60
64	40	30
256	20	15
1024	10	8
4096	5	4
16384	3	2

Tab 1: relazione tra i parametri

Non sono stati considerati ulteriori valori a causa della condizione PPE ≤ 32K=32768; in realtà non si prenderà in considerazione neanche l'ultimo valore di PPE, poiché, come si vedrà nell'algoritmo, il numero di locazioni di memoria necessario è pari a 6*PPE e il numero massimo di locazioni di memoria utilizzabile è 32K=32768 che è minore di 6*PPE=6*16384=98304.

Nel seguito vengono riportati l'algoritmo di conversione (nel caso di PPE=16) con la relativa descrizione e le misure di prestazione.

5.2 Algoritmo di conversione RGB-HSV

In Appendice A è riportato il codice dell'algoritmo

Descrizione dell'algoritmo:

Il programma Assembler inizia con una sezione contenente le direttive per il simulatore: per prima cosa si definiscono delle costanti utilizzate all'interno del programma per mezzo delle direttiva "*#define*".

- Nella prima parte delle definizione si settano i parametri di configurazione:
  1. Si disattiva la configurazione toroidale ponendo "torus=0";
  2. Si pongono 80 PE lungo la direzione orizzontale;
  3. Si pongono 60 PE lungo la direzione verticale;
  4. La memoria locale di ogni processore è di 256 parole da 16 bit;
  5. Il numero dei pixel per ogni processore (PPE) è pari a 16;

     Le immagini in ingresso sono costituite da 320x240 pixel: ponendo 80 PE lungo la direzione orizzontale e 60 lungo quella verticale ogni processore conterrà un'immagine 4x4 coerente mente con il fatto che il PPE è stato fissato a 16.
- Nella seconda parte si definiscono:
  1. IMG_RGB: indica l'indirizzo della locazione di memoria di ogni PEa partire dalla quale si memorizzano le tre coordinate RGB della particolare porzione di immagine.Di fatto funge da puntatore ad un numero di locazioni contigue pari a 3xPPE, in cui il "3" è dovuto alla presenza delle tre coordinate RG B.
  2. IMG_HSV: si riferisce all'indirizzo della locazione di memoria a partire dalla quale si memorizzano le coordinate HSV della porzione di immagine relativa al particolare PE; anche in questo caso la funzione è quella di puntatore a 3xPPE locazioni di memoria contigue.
  3. NUM_PIXEL è pari al PPE –1.

Il programma è costituito dalle quattro funzioni:

- CONVERTI_RGB_HSV
- DIVISIONE
- LOAD_MEM
- MIN_MAX

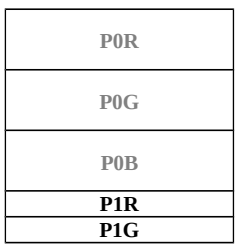
le quali sono localizzate, all'interno del programma, in corrispondenza delle omonime label e fanno tutte parte del file "conversione.spa", per poterle richiamare, quindi, è sufficiente un'istruzione "S_CALL" seguita dal solo nome della funzione e non è necessario specificare il nome del file.

La funzione principale è “CONVERTI_RGB_HSV” ed è quella che effettivamente esegue la conversione RGB➔HSV e richiama le altre tre funzioni. Nel seguito si descriveranno le funzioni nel dettaglio.

5.2.1 Routine LOAD_MEM

Serve per caricare, nella memoria locale di ogni PE, le tre coordinate dei pixel della porzione di immagine da analizzare.
L’algoritmo si basa su due cicli annidati: nel ciclo esterno ogni iterata corrisponde ad una delle tre coordinate RGB, in quello interno ad ogni iterata si fa riferimento ad uno dei pixel della porzione di immagine. E’, quindi, necessario utilizzare due contatori: per il ciclo esterno si è scelto il registro scalare “SR1”, per quello interno “SR0”.
All’interno della funzione si utilizza l’istruzione “SAMPLE”, per mezzo della quale i 16 sensori ottici dei PE campionano l’immagine e i valori vengono memorizzati nei PR (pixel registers). L’istruzione “SAMPLE” legge, in successione, ogniqualvolta viene richiamata, una riga del file di testo “*.blf” in input. I valori memorizzati nei pixel register variano tra 0 e 255 e si riferiscono alla scala di grigi, le immagini in ingresso sono invece a colori: è necessario scomporre l’immagine nelle tre componenti RGB e mandare in input al simulatore i corrispondenti tre file “*.bmp”; a tal fine è sufficiente creare un file “*.blf” in cui nella prima riga si scrive il nome del file “*.bmp”, contenente l’immagine relativa alla componente R, nella seconda quello relativo alla componente G, e nella terza quello relativo alla componente B.
All’interno della funzione l’istruzione “SAMPLE” compare nel ciclo più esterno, quindi verrà eseguita proprio nei tre cicli, una volta per ogni componente.
Una volta nei pixel register, i valori devono essere portati nei registri vettoriali e poi in memoria a partire dalla locazione avente come indirizzo “IMG_RGB”; da questa locazione in poi la memoria contiene le tre coordinate, nell’ordine RGB, dei pixel da 0 a 15, in maniera interleaved.

La configurazione della memoria sarà quella di Fig 5.1 in cui PXY sta per “pixel X coordinata Y” con Y=R, G, B:



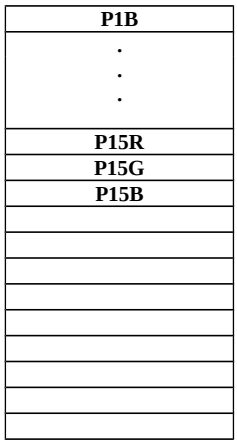


Fig 5. 1: Stato della memoria dopo l'esecuzione della funzione LOAD_MEM

Analisi del codice Assembler:

- Si pone il contenuto del registro SR1 (contatore relativo al ciclo più esterno) uguale a tre per tenere conte delle tre coordinate.
- Si azzera il contenuto del registro vettoriale R3: nella prima iterata vale 0, nella seconda vale 1 e nella terza vale 2 in riferimento, rispettivamente, alle coordinate R G e B.
- Ha inizio il ciclo esterno all'interno del quale si esegue il ciclo più interno un numero di volte pari al PPE per agire sulla singola componente.
- Si carica nel contatore del ciclo interno il valore NUM_PIXEL per contare, a ritroso, le iterate da NUM_PIXEL sino a zero, incluso, per un totale di PPE iterate.
- Nella parte relativa al ciclo interno si caricano in memoria le coordinate del singolo pixel, a partire dall'ultimo pixel: le locazioni di memoria sono contigue e pari a 3xPPE: da IMG_RGB a IMG_RGB+3*PPE-1.
- Nella prima iterata del ciclo più interno si considererà, nelle tre volte del ciclo esterno, rispettivamente:

  IMG_RGB+3*PPE-3

  IMG_RGB+3*PPE-2

  IMG_RGB+3*PPE-1

  che equivale a:

  IMG_RGB+3*NUM_PIXEL+0

  IMG_RGB+3*NUM_PIXEL+1

  IMG_RGB+3* NUM_PIXEL+2

  la differenza nell'utilizzo delle tre coordinate è, quindi, che la prima volta si somma 0, la seconda 1 e la terza 2 a IMG_RGB+3* NUM_PIXEL; questo valore viene salvato nel registro

R3 e viene incrementato ogni volta alla fine del ciclo esterno; il puntatore alla prima coordinata del pixel, cui si riferisce la particolare iterata del ciclo interno, viene, invece, salvata nel registro R1.

- Il ciclo interno inizia caricando nel registro R0 il contenuto del PR il cui indirizzo è salvato nel registro SR0 e che viene incrementato ad ogni iterata del ciclo interno; in R0 c'è, quindi, il valore del pixel: per mezzo dell'istruzione "STORE", viene salvato nella locazione di memoria relativa alla coordinata del pixel, il cui indirizzo è contenuto in R1.
- Prima della fine del ciclo interno si fa in modo che R1 punti alla prima coordinata del pixel precedente e si decrementa il contatore SR0.
- Prima della fine del ciclo più esterno si aggiorna l'offset rispetto alla prima coordinata, si salva nel registro R3 e si decrementa il contatore SR1.

5.2.2 Routine DIVISIONE

Il simulatore non fornisce un'istruzione specifica per l'operazione di divisione quindi, visto che questa ricorre spesso nelle formule di conversione, è stato utile sviluppare una funzione apposita che effettui la divisione *intera*; per una descrizione dettagliata si veda l'Appendice B.

Nell'algoritmo si suppone che il dividendo sia memorizzato in un registro di partenza (R_{d1}), il divisore in un altro registro (R_{d2}) che si utilizzi un registro di supporto (R_s) per il dividendo e che il quoziente sia in R_q.

L'algoritmo simula l'operazione di divisione fatta manualmente: si considera la cifra più significativa del dividendo e si confronta con il divisore; se risulta maggiore o uguale si incrementa il risultato di uno, altrimenti si considerano via via le cifre meno significative del dividendo fino al raggiungimento della precedente condizione. Una volta incrementato il risultato, stando attenti che ogni incremento si deve riferire via via alle cifre meno significative del risultato, si pone il dividendo pari alla differenza tra il valore precedente e il divisore e si effettua il ciclo sopra descritto: questo sino a quando vengono analizzate tutte le cifre del dividendo

Nel caso in questione si tiene presente che in ogni divisione il numeratore è sempre minore o uguale al denominatore e si utilizza questa proprietà per ottimizzare l'algoritmo della divisione (si veda l'appendice per ulteriori approfondimenti)

Analisi del codice Assembler:

R11 funge da registro di supporto

R14 contiene il divisore

R15 contiene il risultato finale

L'algoritmo inizia ponendo il contenuto del registro SR4 uguale a nove, visto che questo funge da contatore per le iterate, e azzerando il valore del registro R15. A questo punto inizia il loop all'interno del quale:

Si pone in R1 la differenza tra R11 ed R14.
Si effettua uno shift a sinistra di un bit del registro contenente il risultato.
Se il contenuto di R1 è minore di 0 (cioè se il contenuto di R11 è minore di quello di R14) il PE entra in uno stato di SLEEP.
Altrimenti si mette 1 come LSB (Less Significant Bit) del risultato e si pone in R11 il contenuto di R1 che era pari alla differenza tra R11 e R14.
Si svegliano gli eventuali PE in stato di SLEEP.
Si decrementa il contatore e se questo è non negativo, cioè se sono state effettuate al più nove iterate, si torna alla label di inizio ciclo.

5.1.3 Routine MAX-MIN

Dati tre valori in ingresso, tale funzione determina sia il massimo che il minimo dell'insieme. Si è preferito sviluppare una funzione che li determini entrambi piuttosto che due separate perché, per la struttura dell'algoritmo, determinato l'uno si trova facilmente anche l'altro.

L'algoritmo procede nel seguente modo:

- si confrontano i primi due valori:

  se il primo minore del secondo

  allora min = primo valore

  max = secondo valore;

  se il terzo valore è minore del primo

  allora min=terzo valore

  se il terzo valore è maggiore del secondo

allora max=terzo valore

altrimenti min = secondo valore

max = primo valore;

se il terzo valore è minore del secondo

allora min=terzo valore

se il terzo valore è maggiore del primo

allora max=terzo valore

Analisi del codice Assembler:

Si suppone che i tre valori da confrontare si trovino nei registri R6, R7 ed R8. E che il minimo si pone nel registro R4 e il massimo nel registro R5.

R1=R6-R7
Tutti i PE per i quali R1≥0, cioè se R6≥R7, entrano in uno stato di SLEEP e si pone nel loro SLEEP VECTOR il valore 1;
R4=R6 e R5 = R7 per i PE operativi

R1=R8-R6
Tutti i PE per i quali R1≥0, cioè se R8≥R6, entrano in uno stato di SLEEP e si pone nel loro SLEEP VECTOR il valore 2;
R4=R8 per i PE operativi e si svegliano i PE con valore 2

R1=R7-R8
Tutti i PE per i quali R1≥0, cioè se R7≥R8, entrano in uno stato di SLEEP e si pone nel loro SLEEP VECTOR il valore 4;
R5=R8 per i PE operativi e si svegliano i PE con valore 4;
Si pongono nello stato di SLEEP tutti i PE ancora svegli, cioè quelli per i quali il primo valore era minore del secondo, e si pone nel loro SLEEP VECTOR il valore 8, e si riattivano quelli per i quali il primo valore era maggiore o uguale al secondo che erano in uno stato di SLEEP con valore 1; si

procede in maniera analoga a quanto fatto precedentemente semplicemente invertendo i ruoli di R6 ed R7.

5.2.4 Routine *CONVERTI_RGB_HSV*

E' la funzione principale all'interno della quale si effettua la conversione vera e propria da coordinate RGB ad HSV

Come già detto, sono state definite due costanti IMG_RGB ed IMG_HSV le quali indicano gli indirizzi delle locazioni di memoria contigue a partire dalle quali vengono salvate, rispettivamente, le coordinate RGB ed HSV dei PPE pixel. Il registro R2 funge da puntatore alla prima zona di memoria e il registro R3 alla seconda.
La funzione è costituita da un loop all'interno del quale si effettua la conversione di coordinate relativamente ad un solo pixel: si effettua un numero di iterate pari PPE; si utilizza il registro SR0 come contatore, si carica al suo interno il valore NUM_PIXEL e inizia il ciclo vero e proprio.
Per ogni pixel si deve determinare la coordinata di valore massimo e quella di valore minimo tra R, G e B; come previsto dalla funzione MIN_MAX, queste vengono poste nei registri R6, R7, R8 tramite le istruzioni "LOAD" e si richiama la apposita funzione la quale porrà in R4 la coordinata di valore minimo ed in R5 quella di valore massimo.
A questo punto è possibile determinare le coordinate V, S ed H.
V:
E' noto che V = max, in cui max è il valore normalizzato rispetto a 255; max può assumere valori tra 0 e 255: la divisione tra max e 255, essendo questa intera dà come risultato 0 oppure 1; questo è coerente con il fatto che V ϵ [0..1] ma è sicuramente poco significativo.
E' utile quantizzare l'intervallo [0..1] in 256 sotto-intervalli e fare in modo che la coordinata V appartenga ad uno dei 256 sotto-intervalli; a tal fine è sufficiente effettuare non la divisione tra max e 255 ma tra (max*256) e 255: il risultato ottenuto è V*256; la moltiplicazione per 256 corrisponde, in aritmetica binaria ad uno shift aritmetico a sinistra di 8, la divisione ad uno shift di 8 a destra: poiché max ϵ [0..255], gli otto bit più significativi saranno nulli ed uno shift a sinistra di otto bit, quindi, non causa perdita di bit. Detto ciò per determinare la coordinata V si dovrebbe effettuare max*256/255; poiché la divisione che si effettua è intera, 256/255 è pari ad uno quindi si può direttamente porre V=max, evitando così l'esecuzione della funzione DIVISIONE che aggiungerebbe un notevole costo computazionale

Si fa puntare R3 alla locazione di memoria prevista per la coordinata H del pixel cui si riferisce la particolare iterata, e si conserva il risultato in memoria per mezzo dell'istruzione "STORE".

S:

E' noto che S=delta/max essendo delta=max-min.

- Si pone in R9 il valore delta.
- Si analizza il caso in cui delta≠0 e max≠0; a tal fine tutti i PE per i quali R9=0 entrano nello stato di SLEEP ponendo 1 nel vettore di SLEEP ; tutti quelli con R5 = 0 vengono posti nello stato di SLEEP con 2.
- Si pongono delta in R11 e max in R14, essendo questi due i registri, all'interno della funzione DIVISONE, che fungono da dividendo e da divisore, rispettivamente.
- Per motivi analoghi a quanto detto a proposito della coordinata V, si effettua la divisione tra delta*256 e max. Viene richiamata la funzione che effettua la divisione ed il risultato si trova in R15; il valore di tale registro viene posto nella locazione di memoria cui puntava R3, cioè quella relativa alla coordinata S del pixel. Tutti i pixel attualmente operativi vengono disattivati; i due insiemi di PE che erano nello stato di SLEEP con valori 1 e 2 nel vettore di SLEEP vengono entrambi risvegliati con l'istruzione "WAKEUPI 3": si pone H=-1 ed S=0; si svegliano tutti i PE.

H:

Si effettua ora il calcolo della H per mezzo della formula approssimata di Smith.

Tale coordinata viene calcolata con tre formule di conversione diverse a seconda del fatto che la coordinata con valore massimo sia R, G o B.

- Si vede se R è la coordinata di valore massimo confrontando il contenuto del registro relativo alla coordinata R e quello contenente il valore massimo.
- Si fanno entrare nello stato di SLEEP tutti i PE per i quali questo non è vero ponendo 1 nel vettore di SLEEP.
- Si pone nel registro R11, che contiene il dividendo, la differenza tra la coordinata G e B moltiplicata per 256 per i motivi di cui si è detto precedentemente; in R14 si pone il valore del dividendo che in questo caso è delta. La funzione di divisione sviluppata fornisce risultati corretti soltanto se sia dividendo che divisore sono positivi. Nel caso in questione, il divisore è delta ed è sicuramente non negativo; il dividendo, invece, è G-B e può essere negativo. In tal caso si moltiplica il dividendo per –1, rendendolo positivo e si pone 1 nel registro R12, che funge da flag.
- Si effettua normalmente la divisione tra due numeri positivi e si pone il risultato, non maggiore di 256, nel registro R15.

- Si moltiplica il risultato per 60 e, dati i valori in gioco, si è sicuri di ottenere un valore a 16 bit e quindi interamente contenuto nel registro R15. Poiché il numeratore era stato moltiplicato per 256 anche il risultato ha un fattore 256 in più, quindi si divide il risultato per 256 semplicemente con uno shift aritmetico e destra di 8 bit.
- Si deve ancora verificare se il risultato della divisione deve essere positivo o no: si vede il valore di R12 se questo è pari ad 1 si moltiplica per –1 il risultato e si somma 360: questo per assicurare che il valore della coordinate H ϵ [0..360]. Si azzera il contenuto del registro R12 così da sfruttare correttamente in seguito, e si memorizza il risultato finale, contenuto in R15, nella locazione di memoria puntata da R13. Entrano nello stato di SLEEP con il valore 2 tutti i PE svegli cioè quelli per i quali R=max.
- Si riattivano tutti i PE per i quali R non era la coordinata massima il cui SLEEP vector conteneva 1.
- Analogamente a prima si vede se G é la coordinata di massimo valore; se no questo significa che la coordinata di massimo valore è B, e tali PE entrano nello stato di SLEEP con il valore 1; in tal caso H=(2+(B-R)/delta)*60. Anche in questo caso si moltiplica per 256 il dividendo per i motivi prima esposti; effettuo la divisione tra (B-R)*256 e delta; chiaramente poi si dovrà sommare 2*256=512. Anche in questo caso B-R può essere negativi e quindi si moltiplica questa quantità per –1 e si pone 1 in R12, che funge da flag.
- Si effettua la divisione, eventualmente, si cambia di segno il risultato e si somma 512. Si ha per ora in R15 la quantità 2+(B-R)/delta, sicuramente positiva, moltiplicata per 256. In R15 si potrebbe avere, al limite, 3*256=768; a questo punto si deve effettuare la moltiplicazione per 60, come dalla formula, al limite si avrà 768*60=46080: questo valore non può essere contenuto interamente in R15 essendo questo a 16 di bit di cui 1 bit per il segno: può contenere al massimo 2^{15} =32768: si deve effettuare la moltiplicazione che prevede un risultato a 32 bit e che utilizza l’accumulatore.
- Si azzera l’accumulatore e si effettua la moltiplicazione con l’istruzione “MACC”: si avrà la parte alta del risultato in R0 e la parte bassa in R15; il risultato ottenuto è moltiplicato per 256; si dovrebbe effettuare una divisione per 256, quindi una rotazione del complesso dei due registri R0-R15 a destra di otto bit: poiché gli otto bit più significativi di R0 sono sicuramente nulli, il risultato complessivo potrà essere interamente contenuto nel registro R15 e a tal fine si effettua uno shift e destra di otto bit di R15, uno e sinistra di otto bit di R0 e si sommano, in R15, i contenuti dei due registri: R15 contiene il risultato definitivo per la coordinata H che viene memorizzata nelle locazione di memoria puntata del registro R3.

- Si riattivano tutti i PE per i quali B era la coordinata massima il cui SLEEP VECTOR contiene il valore 1. In tal caso H=(4+(R-G)/delta)*60, si effettua la divisione tra (R-G)*256 e delta per i motivi prima sottolineati; poi si dovrà sommare 4*256=1024. Anche in questo caso R-G può essere negativo e quindi si moltiplica questa quantità per –1 e si pone R12 pari ad 1. Analogamente a prima si ottiene in R15 la quantità 4+(R-G)/delta, sicuramente positiva, moltiplicata per 256. In R15 si potrebbe avere, al limite, 5*256=1280; effettuando la moltiplicazione per 60 si potrebbe avere 1280*60=76800. Analogamente a prima, quindi, si deve effettuare la moltiplicazione che prevede un risultato a 32 bit e che utilizza l'accumulatore. Con la stessa tecnica utilizzata nel caso in cui era G=max, R15 conterrà il risultato definitivo per la coordinata H la quale verrà memorizzata nelle locazione di memoria puntata del registro R3.
- Si è quindi giunti alla fine del ciclo: si aggiornano i due puntatori alle due locazioni di memoria, cioè R2 e R3, si decrementa il contatore SR0 e si vede se effettuare altre iterate o se si è giunti alla fine.

5.3 Misure di prestazione

Nel seguito si illustreranno alcuni diagrammi relativi a vari tipi di misure di prestazioni e relativi risultati.

5.3.1Istogramma dinamico

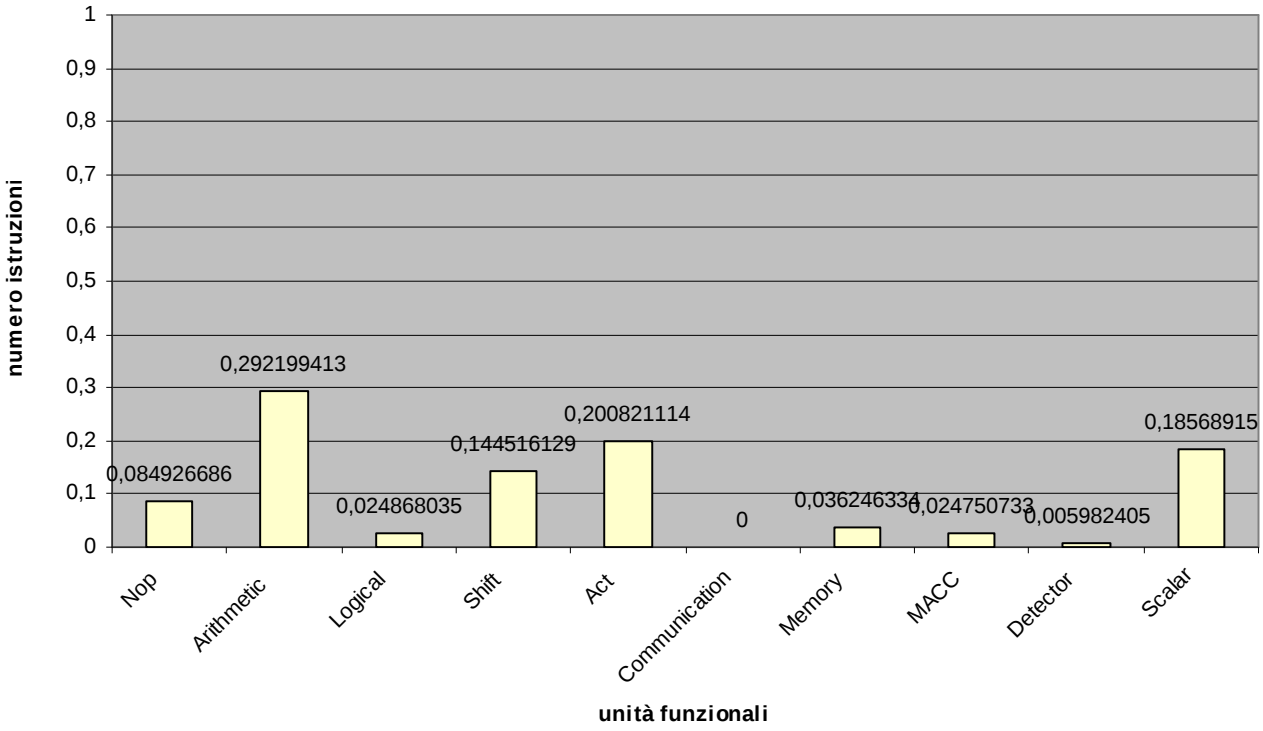


Fig 5. 2: Istogramma dinamico

Il diagramma di Fig 5.2 mostra l'utilizzo delle varie unità funzionali, poste in corrispondenza con le classi di istruzioni dell' Istruction Set di SIMPil: ad ogni classe corrisponde un'unità funzionale e ogni barra verticale indica la parte di istruzioni della particolare classe rispetto al totale di istruzioni emesse.

Si vede che, come spesso accade, una buona parte delle istruzioni è di tipo aritmetico; si ha anche un notevole numero di istruzioni scalari che indica la percentuale di parte sequenziale dell'algoritmo.

Una nota negativa è data dall'elevato valore delle istruzioni di *Activity*: in una notevole porzione del totale dei cicli macchina i PE sono in stato di SLEEP e questo, chiaramente, influisce negativamente sul throughput finale.

Quanto detto si riferisce al caso di PPE=16.

Viene riportata una tabella che mostra per gli altri valori del PPE il numero totale di istruzioni delle relative classi, il totale di istruzioni vettoriali e di istruzioni scalari:

PPE	1	4	16	64	256	1024	4096
NOP	47	183	724	2883	11524	46083	184323
ARITHMETIC	164	631	2491	9931	39691	158731	634891
LOGICAL	17	56	212	836	3332	13316	53252
SHIFT	77	308	1232	4928	19712	78848	315392
ACT	107	428	1712	6848	27392	10956	438273
COMMUNICATION	0	0	0	0	0	0	0
MEMORY	24	81	309	1221	4869	19461	77829
MACC	13	55	211	835	3331	13315	53251
DETECTOR	6	15	51	195	771	3075	12291
VETTORIALI	455	1757	6942	27677	110622	442379	1769502
SCALARI	101	408	1583	6287	25103	100367	401423

Tab. 2: Distribuzione della classi di istruzioni vettoriali e scalari al variare del PPE

5.3.2 Analisi per PPE variabile

Come già detto, l'immagine da partizionare può essere divisa in sotto-immagini in modo diversi a seconda del numero di pixel relativo ad ogni processore (PPE).

Si può studiare l'andamento dell'utilizzo del sistema e del throughput al variare del PPE.
Si tenga presente che:

$$U = \frac{R_I}{N_I * N_P};$$

$$TH = \frac{N_I * N_P * U}{T_{EX}}$$

$$T_{EX} = \frac{N_I}{f_{CK}}$$

in cui:

U = utilizzo del sistema

TH = throughput di sistema

R_I = totale di istruzioni eseguite da tutti i processori tra quelle emesse

N_I = totale di istruzioni emesse per il singolo processore

N_P = numero totale di processori

T_{EX} = tempo di esecuzione

f_{ck}= frequenza di clock pari a 500MHz

i valori R_I e N_I sono forniti dal simulatore

I valori utilizzati per tali parametri sono quelli riportati in tabella:

PPE	Memoria		Cicli	Utilizzo sistema	Tempo di esecuzione	Throughput
	PE [byte]	***System*** [Kb]	**Totali**			
1	12	28.125	556	0.514788	2.912E-14	6.17746E+20
4	48	28.125	2165	0.522125	1.12448E-13	1.56637E+20
16	192	28.125	8525	0.516060	4.44288E-13	3.87045E+19
64	768	28.125	33964	0.514614	1.77133E-12	9.64901E+18
256	3072	28.125	135725	0.514227	7.07981E-12	2.41044E+18
1024	12288	28.125	542764	0.507034	2.83134E-11	6.33793E+17
4096	49152	28.125	2170925	0.507018	1.13248E-10	1.58443E+17

Tab 3: Confronto di prestazioni al variare del PPE

PE rappresenta il numero di parole di memoria da 16 bit occupate. Si è già detto che PE=PPE*6

System è pari a PP per il numero di processori, quindi è proporzionale a PPE*X_SIZE*Y_SIZE;

quest'ultimo è un valore costante, infatti il valore di System è lo stesso al variare del PPE.

In Fig 5.3 e Fig 5.4 vengono riportati l'istogramma e il diagramma lineare dell'utilizzo di sistema in funzione del PPE:

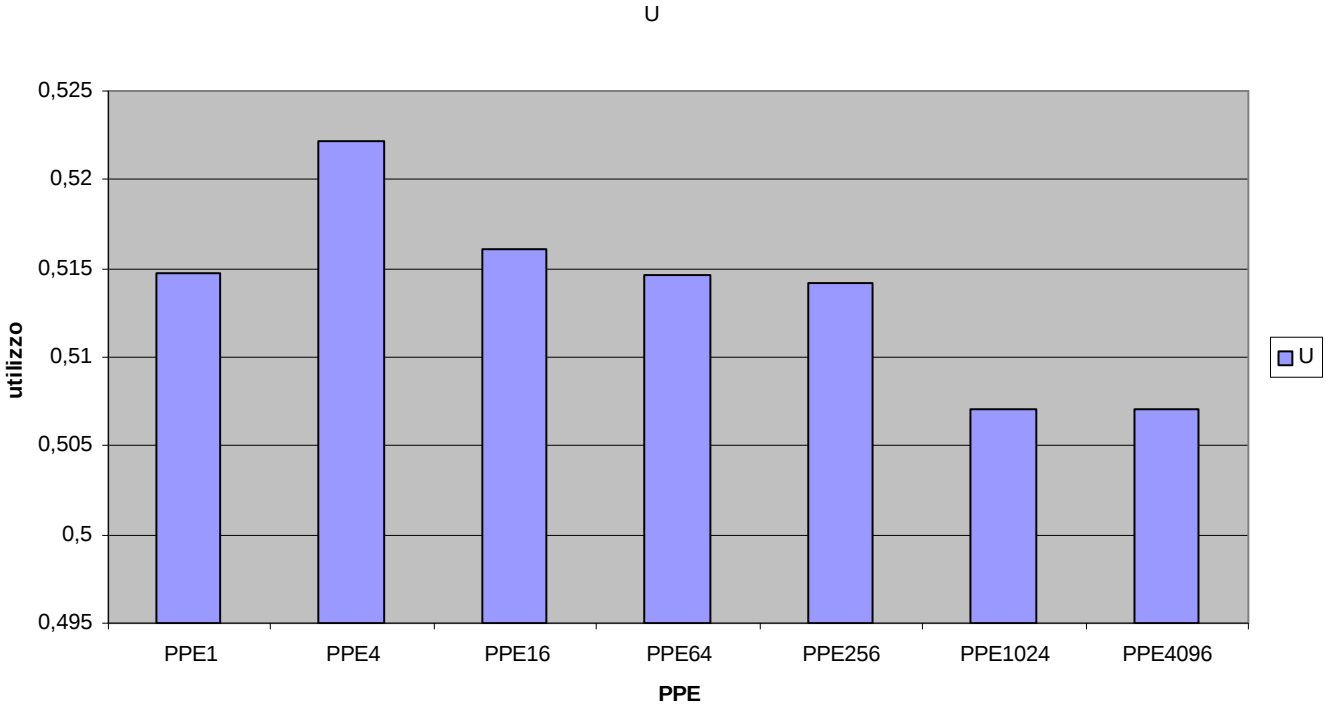


Fig 5. 3: Istogramma dell'utilizzo di sistema

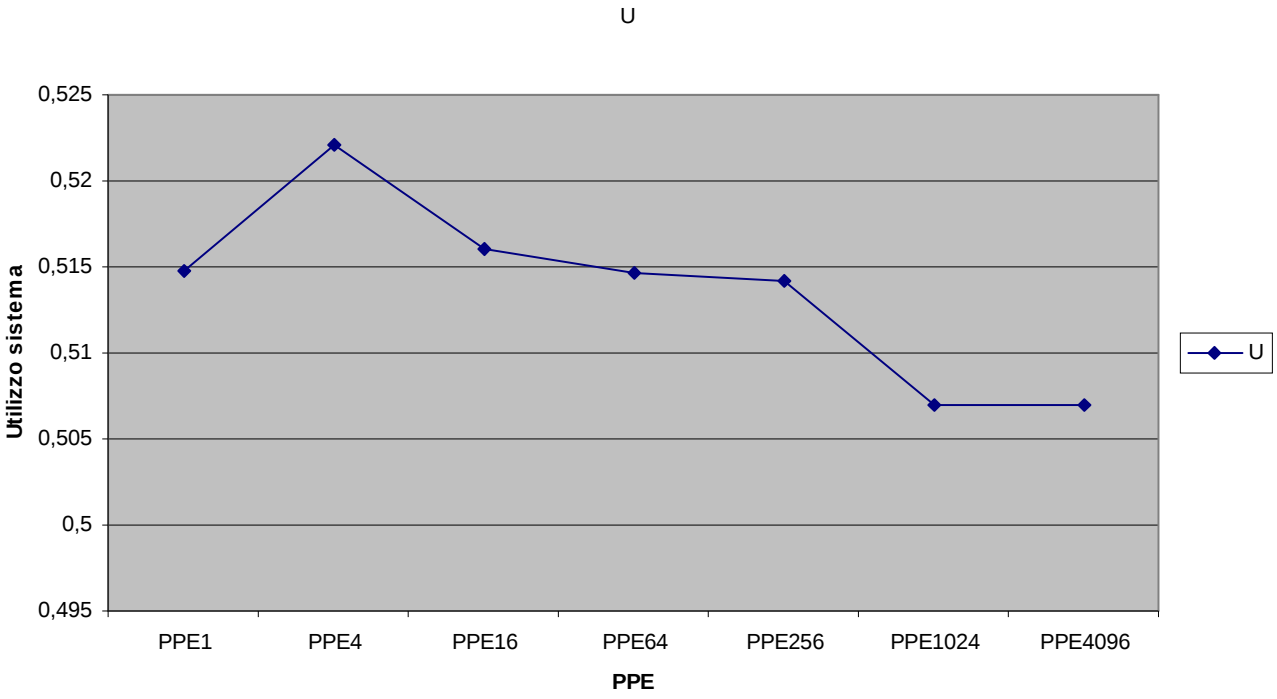


Fig 5. 4: Diagramma lineare dell'utilizzo di sistema

Come si vede il valore di PPE che ottimizza l'utilizzo di sistema è 4.

In Fig 5.5 viene riportato il diagramma del throughput al variare del PPE; data la grande differenza tra i valori del throughput nei vari casi, sull'asse delle ordinate è stata utilizzata una scala logaritmica:

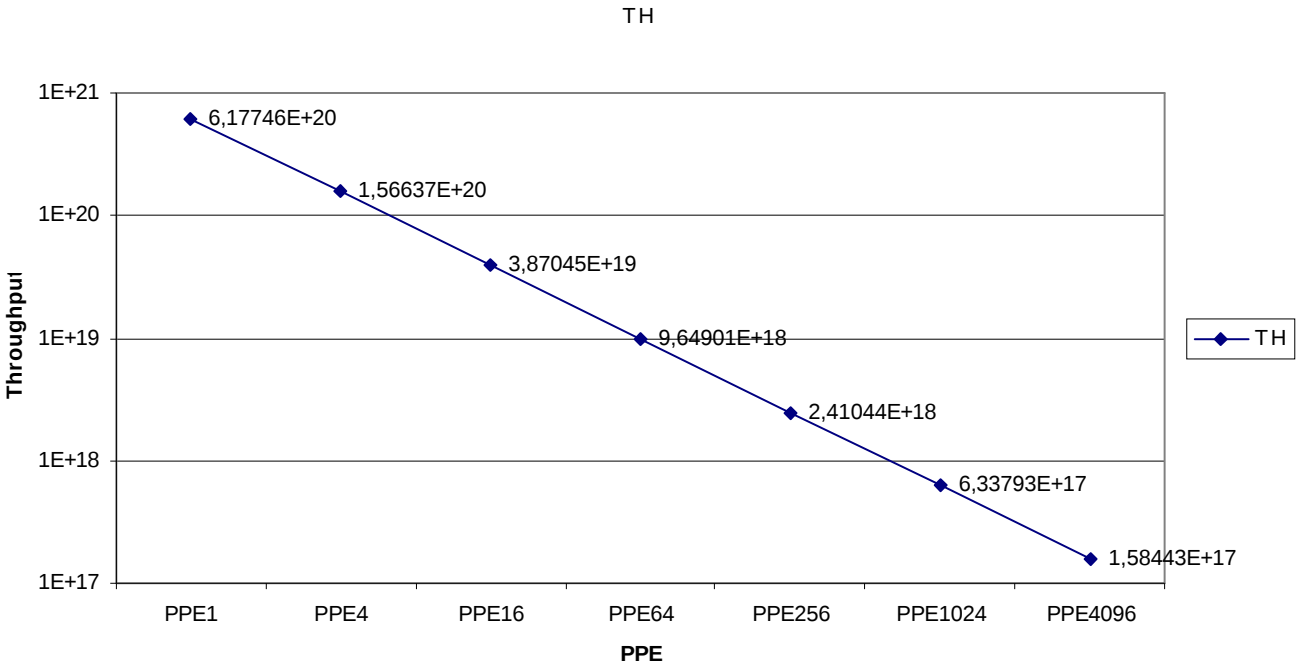


Fig 5. 5: Diagramma logaritmico del throughput di sistema

Si vede come il throughput decresce molto velocemente all'aumentare del PPE; d'altronde questo è coerente con il fatto che in tal modo diminuisce il grado di parallelismo.

5.3.3 Costo della divisione

Una considerevole parte dell'esecuzione dell'algoritmo, come viene spiegato nell'appendice B, viene impiegato nell'esecuzione della divisione; oltre ad ottimizzare l'algoritmo stesso, si può vedere quali risultati si otterrebbero se si utilizzasse un'implementazione hardware del divisore: invece delle 95 istruzioni che costituiscono la funzione divisione si dovrebbe eseguire una sola istruzione, ottenendo un risparmio nel numero totale di istruzioni.

In Fig 5.6, si rappresenta l'andamento del risparmio nel numero di istruzioni totali in presenza di divisore (numero totale istruzioni senza divisore- numero totale istruzioni con divisore) rispetto al numero totale di istruzioni (numero totale istruzioni senza divisore).

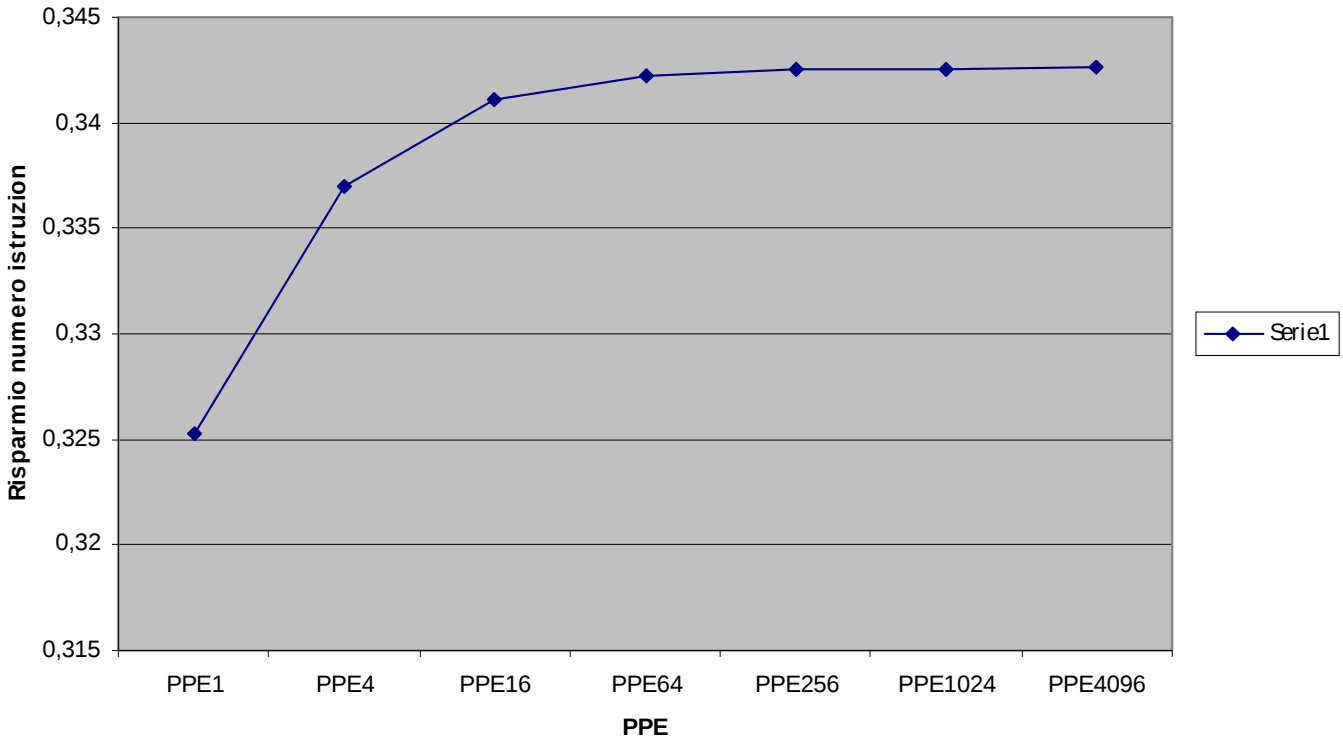


Fig 5. 6: Diagramma risparmio del numero di istruzioni in presenza di una implementazione hardware del divisore rispetto al numero totale di istruzioni

Si vede che all'aumentare del PPE il risparmio è sempre maggiore ma da PPE=256 in poi rimane praticamente costante.

Si noti, però che, se da un lato aumenta il numero di istruzioni emesse per unità di tempo dall'altro la presenza del divisore fa aumentare le dimensioni dei processori e questa ha conseguenze negative per ciò che riguarda l'efficienza d'area

Appendice A

```
#define _torus 0
#define _x_size 80
#define _y_size 60
#define _LOCAL_MEM_SIZE 256
#define _ppe 16



#define           IMG_RGB           0
#define           IMG_HSV           48
#define           NUM_PIXEL         15

;R1=registro di comodo

;R2=registro che punta alla locazione di memoria che punta
;alla particolare coordinata RGB

;R3=registro che punta alla locazione di memoria che punta
;alla particolare coordinata HSV

;R4=min     R5=max      R6=R  R7=G  R8=B


CONVERTI_RGB_HSV  NOP

                  S_CALL      LOAD_MEM

                  LOADI IMG_RGB
                  ADDI  R2,R0,0

                  LOADI IMG_HSV
                  ADDI  R3,R0,0


                  S_LOADI     NUM_PIXEL   ;num totale di pixel-1


LBL               NOP

;ad ogni iterata del ciclo si convertono le coordinate RGB di 1 pixel in
;coordinate HSV

                  LOAD  R6,R2
                  ADDI  R2,R2,1           ;R6=Ri

                  LOAD  R7,R2
                  ADDI  R2,R2,1           ;R7=Gi

                  LOAD  R8,R2             ;R8=Bi

                  S_CALL      MIN_MAX

                  ADDI  R3,R3,2           ;punta alla coordinata Vi


                  STORE R3,R5       ;*Vi=maxi
                  SUBI  R3,R3,2     ;punta alla coordinata Hi
```

```
                SUB   R9,R5,R4    ;delta=max-min
                SEQI  R9,0x1
                SEQI  R5,0x2


;sono operativi quelli per i quali R9!=0 e R5!=0; delta!=0 e max!=0

                ADDI R3,R3,1            ;punta alla coordinata S_i
                ADDI  R11,R9,0

                ADDI  R14,R5,0


;R15=R13/R14; R3=R9-R5 ; Si=delta-max

                S_CALL      DIVISION



                STORE R3,R15            ;*S==delta:max
                SUBI  R3,R3,1           ;punta ad H_i

;si rendono inattivi tutti quelli attualmente operativi

                XOR   R1,R1,R1
                SEQI  R1,0x4

;si riattivano tutti quelli per i quali delta=0 e quelli per i quali max=0

                WAKEUPI     0x3

                LOADI -1
                STORE R3,R0             ;*H_i=-1
                XOR   R0,R0,R0
                ADDI  R3,R3,1           ;R3 punta a S_i
                STORE R3,R0             ;*S_i=0
                SUBI  R3,R3,1           ;punta ad H_i
                WAKEUPI     0xFF        ;tutti svegli

;si effettua il calcolo della H secondo SMITH

                SUB   R1,R6,R5          ;R1=R_i-max
                SNEI  R1,0x1

                SUB   R11,R7,R8

                SGEI  R11,0x20

                MULI  R11,R11,-1


                XOR   R12,R12,R12
                ADDI  R12,R12,1
                WAKEUPI     0x20


                ADDI  R14,R9,0          ;R13=delta

;R15=R13/R14
```

```
                    S_CALL      DIVISION


                    MULI  R15,R15,60
                    LSHI  R15,R15,-8



                    SEQI  R12,0x20

                    MULI  R15,R15,-1

                    LOADI 360
                    ADD   R15,R15,R0

                    WAKEUPI     0x20
                    XOR   R12,R12,R12




                    STORE R3,R15            ;*Hi=(g-b):delta
                    SUB   R1,R6,R5          ;R1=Ri-max
                    SEQI  R1,0x2            ;se Ri=max etra nello stato di SLEEP
                    WAKEUPI 0x1

;si riattivano tutti quelli che non hanno R=max



                    SUB   R1,R7,R5    ;R1=Gi-max
                    SNEI  R1,0x1      ;erano nello stato di SLEEP quelli con R=max
                                      ;ora entrano stato di SLEEP quelli che
                                      non ;hanno G=max quindi hanno B=max


                    SUB   R11,R8,R6



                    SGEI  R11,0x20

                    MULI  R11,R11,-1

                    XOR   R12,R12,R12
                    ADDI  R12,R12,1
                    WAKEUPI     0x20



                    ADDI  R14,R9,0    ;R14=delta

;R15=R13/R14

                    S_CALL      DIVISION

                    SEQI  R12,0x20
                    MULI  R15,R15,-1

                    WAKEUPI     0x20
                    LOADI 512               ;2*2^8
```

```
                    ADD   R15,R15,R0


                    ZACC
                    LOADI 60
                    MACC  R15,R15,R0
                    LACCH R0
                    LSHI  R15,R15,-8
                    LSHI  R0,R0,8
                    ADD   R15,R15,R0

                    XOR   R12,R12,R12

                    STORE R3,R15            ;*Hi=(b-r):delta+2
                    SUB   R1,R7,R5          ;R1=Gi-max
                    SEQI  R1,0x4            ;entrano nello stato di SLEEP
                                           quelli ;che hanno Gi=max

                    WAKEUPI 0x1

;si riattivano tutti quelli che non hanno nè G=max nè R=max

                    SUB   R11,R6,R7   ;R13=R-G


                    SGEI  R11,0x20
                    MULI  R11,R11,-1
                    XOR   R12,R12,R12
                    ADDI  R12,R12,1
                    WAKEUPI     0x20

                    ADDI  R14,R9,0    ;R14=delta

;R15=R13/R14

                    S_CALL      DIVISION



                    SEQI  R12,0x20
                    MULI  R15,R15,-1

                    WAKEUPI     0x20
                    LOADI 1024        ;4*2^8
                    ADD   R15,R15,R0


                    ZACC
                    LOADI 60
                    MACC  R15,R15,R0
                    LACCH R0
                    LSHI  R15,R15,-8
                    LSHI  R0,R0,8
                    ADD   R15,R15,R0


                    XOR   R12,R12,R12

                    STORE R3,R15            ;*Hi=(r-g):delta+4
                    WAKEUPI     0xFF        ;tutti operativi

                    ADDI  R2,R2,1
```

```
		ADDI	R3,R3,3

		S_SUBI		SR0,SR0,1
		S_BGE SR0,LBL


		S_BRA END


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

DIVISION	NOP



		S_XOR SR4,SR4,SR4
		S_ADDI		SR4,SR4,9


		XOR	R15,R15,R15

LOOP_DIV	NOP



		SUB	R1,R11,R14		;R1=R11-R14
		LSHI	R15,R15,1		;shift del quoziente
		SLTI	R1,0x10
		ADDI	R15,R15,1
		ADDI	R11,R1,0		;sostituzione della parte alta del quoziente
					;con la differenza
		WAKEUPI		0x10
		LSHI	R11,R11,1		;shift a sinistra di un bit
		S_SUBI		SR4,SR4,1
		S_BGT SR4,LOOP_DIV



		S_RETURN



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

LOAD_MEM	NOP

	S_XOR SR1,SR1,SR1
	S_ADDI		SR1,SR1,3		;SR1=3
	XOR	R3,R3,R3


LBL1	LOADI NUM_PIXEL			;R0<= NUM_PIXEL

	;S_XOR		SR0,SR0,SR0
	S_LOADI		NUM_PIXEL		;SR0<=NUM_PIXEL



	MULI	R0,R0,3			;serve per considerare le tre coordinate
```

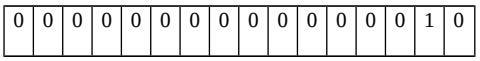
```
      XOR   R1,R1,R1
      ADDI  R1,R0,IMG_RGB           ;R1<= BMPSTART+NUM_PIXEL
                                    ; i pixel so memorizzati da
                                    ;IMG_RGB+NUM_PIXEL a IMG_RGB
                                    ; dal basso verso l'alto
                                    ; l'offset delle tre coordinate (R,G,B)
                                    ;rispetto alla prima
                                    ;vale rispettivamente  (0,1,2)

      ADD   R1,R1,R3
      SAMPLE


LBL2  NOP
      PLOADS      R0,SR0            ;R0<=P(SR0)
      STORE R1,R0                   ;*(R1)<=R0
      SUBI  R1,R1,3                 ;R1=R1-3

      S_SUBI      SR0,SR0,1         ;SR0--
      S_BGE SR0,LBL2          ;si decrementa il contatore relativo al
                        ;numero dei pixel


      ADDI  R3,R3,1
      S_SUBI      SR1,SR1,1
      S_BGT SR1,LBL1


      S_RETURN



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

MIN_MAX     NOP   NOP


      SUB   R1,R6,R7    ;R1=R-G
      SGTI  R1,0x1            ;se R>G dormi
                        ;i PE svegli sono quelli con R<=G


      ADDI  R4,R6,0           ;R4=min=R
      ADDI  R5,R7,0           ;R5=max=G

      SUB   R1,R8,R6    ;R1=R8-R6=B-R
      SGTI  R1,0x2
      ADDI  R4,R8,0           ;R4=min=B
      WAKEUPI     0x2


      SUB   R1,R7,R8    ;R1=R7-R8=G-B
      SGTI  R1,0x4
      ADDI  R5,R8,0           ;R5=max=B
      WAKEUPI     0x4

;entrano nello stato di SLEEP tutti quelli attualmente operativi


      XOR   R1,R1,R1
      SEQI  R1,0x8
```

```
      WAKEUPI     0x1

;sono operativi quelli con G<R

      ADDI  R4,R7,0           ;R4=min=G
      ADDI  R5,R6,0           ;R5=max=R

      SUB   R1,R8,R7          ;R1=R8-R7=B-G
      SGTI  R1,0x2
      ADDI  R4,R8,0           ;R4=min=B
      WAKEUPI     0x2

      SUB   R1,R6,R8          ;R1=R6-R8=R-B
      SGTI  R1,0x4
      ADDI  R5,R8,0           ;R5=max=B
      WAKEUPI     0xFF



      S_RETURN


END   NOP
```

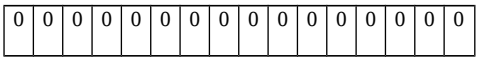
Appendice B

Lo scopo di questa sezione è quello di analizzare nel dettaglio l'algoritmo della divisione [13].
Si suppone che il dividendo sia contenuto in un registro di partenza (R_{d1}), il divisore nel registro (R_{d2}), che si utilizzi un registro di supporto (R_s) per il dividendo e che il quoziente sia in R_q.
Una possibile configurazione iniziale è del tipo:

R_s R_{d1}



R_{d2}



R_q

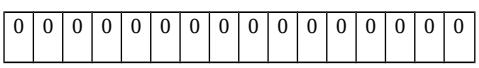
L'algoritmo simula l'operazione di divisione tradizionale:

- si considera il MSB (Most Significant Bit) del dividendo si effettua uno shift a sinistra di un bit, e si pone come LSB (Less Significant Bit) nel registro di supporto
- se l' LSB è maggiore o uguale del dividendo:
  allora si incrementa il risultato di uno e nel registro di supporto si pone la differenza tra il valore precedente e il dividendo.
  altrimenti si considerano via via i bit meno significativi del dividendo, per mezzo di shift successivi sino a quando il contenuto del registro di supporto risulta maggiore o uguale al dividendo
- si effettua uno shift del risultato e si continua come sopra.
- tutto ciò deve essere effettuato all'interno di un loop di 16 iterate: una per ogni bit del registro contenente il dividendo.

Il valore del registro di supporto, del dividendo e del risultato relativo alla sequenza dei passi per l'esempio precedente è di seguito riportata:

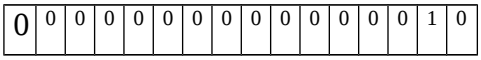
Passo 1:

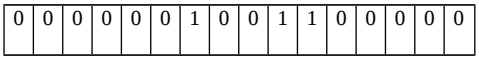
R_s R_{d1}



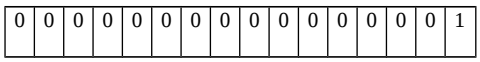
R_q

Passo 2:



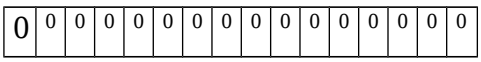


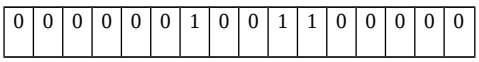
R_s R_{d1}



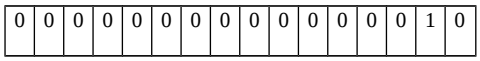
R_q

E successivamente:





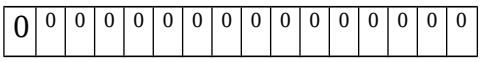
R_s R_{d1}

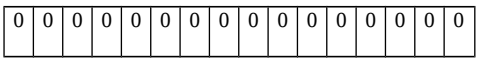


R_q

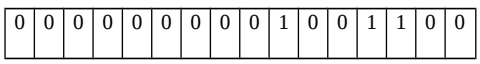
.
.
.

Passo 16:





R_s R_{d1}



R_q

Alla fine il registro di supporto conterrà il resto della divisione intera e il registro che inizialmente conteneva il dividendo sarà pari a zero, visto che in ognuna delle 16 iterata se effettua uno shift aritmetico.
Considerando le formule di conversione approssimate, si vede che la divisione viene effettuata 4 volte. Si vede che tre divisioni si riferiscono, nel calcolo della coordinata H, ai casi in cui il max sia R, G o B: per ogni pixel quindi il processore sarà attivo soltanto in una di queste divisioni e nella esecuzione delle altre due sarà nello stato di SLEEP provocando una notevole perdita nell'utilizzo dei cicli macchina. E', quindi, necessario snellire il più possibile l'algoritmo cercando di ridurre il numero di iterate necessario.
Analizzando le formule di trasformazione si vede che numeratore e denominatore hanno, al più, gli 8 bit meno significativi non nulli:

R_{d1}

0	0	0	0	0	0	0	0	x	x	x	x	x	x	x	x

per i motivi illustrati nella parte riguardante la funzione CONVERTI_RGB_HSV, si effettua la divisione tra (numeratore*256) e denominatore, il numeratore è ruotato e sinistra di otto bit e gli altri bit sono uguali a zero:

x	x	x	x	x	x	x	x	0	0	0	0	0	0	0	0

R_{d1}

Si nota che in tutti i casi in cui la divisione viene richiamata il numeratore, non moltiplicato per 256, è minore o uguale al denominatore. L'algoritmo descritto precedentemente consiste nell'inserire via via i bit più significativi nel registro di supporto fino a quando il suo contenuto è maggiore o uguale al dividendo: poiché il valore del dividendo è minore del divisore e poiché il dividendo risulta ruotato di otto bit, sicuramente nelle prime 7 iterate l'unica azione sarà quella di inserire i bit nel registro di supporto:

Configurazione iniziale:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

R_s

x	x	x	x	x	x	x	X	0	0	0	0	0	0	0	0

R_{d1}

Passo 7:

R_s R_{d1}

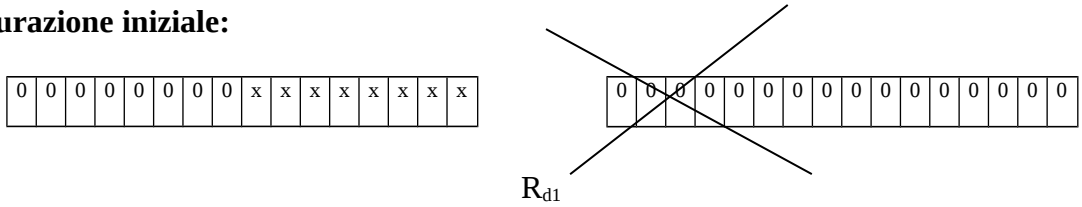
invece di avere il dividendo in un registro iniziale, moltiplicarlo *256 ed inserire per sette iterate i bit nel registro di supporto conviene iniziare supponendo che gli otto bit del divisore siano già nel registro di supporto ed iniziare a confrontare con il dividendo con il divisore con il solito algoritmo, risparmiando sette iterate, cioè quasi dimezzando il numero delle iterate:

Configurazione iniziale:

R_s R_{d1}

Con tale artifizio il dividendo è tutto nel registro di supporto, quindi non è più necessario avere due registri: se ne considera soltanto uno in cui il dividendo è quello di partenza:

Configurazione iniziale:



R_s R_{d1}

RIFERIMENTI BIBLIOGRAFICI

[1] all. ing. Pollacia Giorgio, relatore Prof. F. Sorbello, correlatore ing. S. Vitabile, A.A.1999/2000 "**Tecniche di Analisi del Colore e delle Forme per il Riconoscimento di Segnali Stradali in Contesti Reali**".

[2] A. Gentile, J. Cruz-Rivera, D.S.Wills, L. Bustello, J.J. Figueroa, J.E.Finseca-Camacho, W.E. Lugo-Beauchamp, R.Olivieri, M. Viera-Vera "**Real-Time image Processing on a Focal Plane SIMD Array, in Parallel and Distributed Processing**" Lecture notes in Computer Science (Eds. Jose' Rolim et al.), v. 1586, pp. 400-405, Springer Verlag, New York, 1999.

[3] A. Gentile, A. López-Lagunas, S. M. Chai, H. H. Cat, K. S. Chung, L. Codrescu, M. Deb, S. J. Ryu, M. F. Wolff, J. C. Eble III, W. H. Robinson III, T. Taha, D.S. Wills, M. Viera-Vera, W. E. Lugo-Beauchamp, L. Bustelo, R. Olivieri, J.Figueroa, J. L. Cruz-Rivera, "**Portable Multi-Media Supercomputing**", IEEE Transactions on Computers, 43 pages, 2000.

[4] A. Gentile, H. H. Cat, D. S. Wills, "**The SIMD PIXEL PROCESSOR (SIMPil): SIMPil16 microarchitecture, instruction set and programming model**", Technical Report PICA-TR-1997-15, May 1997.

[5] A. Gentile, H. H. Cat, D. S. Wills, "**The SIMD PIXEL PROCESSOR (SIMPil): SIMPil16 instruction level simulator for windows**", Technical Report PICA-TR-1997-18, June 1997.

[6] SIMPil Home Page, http://www.ee.gatech.edu/research/pica/simpil

[7] S. Vitabile, G. Pollaccia, G. Pilato, F. Corbello, "**Road signs recognition using a dynamic pixel aggregation technique in the HSV color space**".

[8] A. Ford, A. Roberts, "**Colour Space Conversions**", August 11, 1998.

[9] Ohba, K.; Sato, Y.; Ikeuchi, K., "**Visual learning and object verification with illumination invariance**", Proc. of the IEEE/RSJ International Conference on Intelligent Robots and Systems, 1997, Volume: 2 , Page(s): 1044 -1050 vol.2.

[10] Tseng, D.-C.; Chang, C.-H.; "**Color segmentation using perceptual attributes**" Pattern Recognition, 1992. Vol.III. Conference C: 11th IAPR International Conference on Image, Speech and Signal Analysis, pp. 228 – 231.

[11] L. Priese, Jens Klieber, Raimund Lakmann, V. Rehrmann, and Rainer Schian. "**New results on traffic sign recognition**". In IEEE Proc. Intelligent Vehicles'94 Symposium, pages 249--253, 1994.

[12] A. Gentile, H. H. Cat, F. Kossentini, F.sorbello, D. S. Wills, "**Real-Time Vector Quantitazion-based Image Compression on the SIMPil Low Memory SIMD Architecture**", Proc. of 1997 IEEE International Performance, Computing and Communication Conference, Phoenix/Tempe, Arizona, pp. 10-16,1997.

[13] Marlyn Quiñones-Cerpa, Iomar Vargas-Gonzales, Javier E. Fonseca-Camacho, J. L. Cruz-Rivera, "**Parallelization of Spatial Filters on the SIMPil Architecture**".