ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

CORSO DI LAUREA TRIENNALE IN
INGEGNERIA INFORMATICA

Progettazione di filtri di convoluzione riconfigurabili per sistema embedded

Tesi di Laurea sperimentale in **Ingegneria Informatica**

CANDIDATO
Luca Bonfiglioli

RELATORE
Prof. Stefano Mattoccia

SESSIONE II

ANNO ACCADEMICO 2016-2017

Indice

Capitolo 1 - Introduzione

1.1 - Obiettivi

La seguente tesi sperimentale si prefigge l'obiettivo di estendere le funzionalità di un sistema di elaborazione di immagini embedded preesistente (Mattia Bernasconi [1]), introducendo un sistema di configurazione remoto, che utilizza una connessione TCP.

Il sistema di elaborazione si compone di una entità server, costituita da una ZedBoard [4] dotata di un sensore di immagine digitale OV7670 [5] che acquisisce immagini e le elabora mediante un filtro di convoluzione. Il risultato di tale elaborazione viene mostrato su uno schermo VGA e inviato al client come stream di immagini. Oltre allo stream di immagini elaborate, il client riceve i componenti di luminanza e crominanza dell'immagine originale. Tale client, disponibile per Windows, Linux e Mac, trattato nella tesi di Andrea Boscarino [2], si occupa di ricevere le immagini e mostrarle sullo schermo del computer su cui è in esecuzione, ricostruendo le immagini a colori RGB partendo dalle componenti di luminanza e crominanza.

Come accennato, lo scopo di questa tesi è quello di estendere le funzionalità del sistema introdotto precedentemente, introducendo la possibilità di configurare dinamicamente i pesi del filtro (prima limitati a 8 configurazioni predefinite), apportando eventuali ottimizzazioni e valutando i costi in termini di risorse e performance che la configurabilità comporta.

Le modifiche al progetto preesistente interessano:

- Hardware. È infatti necessario che il filtro di convoluzione possa ricevere i parametri di configurazione a tempo di esecuzione e modificare il proprio comportamento dinamicamente.
- Software lato board. Devono essere fornite astrazioni che consentano di configurare il filtro direttamente dal Processing System (ARM) con maggiore semplicità, indipendentemente dall'implementazione hardware del filtro.
- Software lato cliente. L'utente potrà modificare la configurazione del filtro utilizzando un'applicazione client apposita.

1.2 - Contesto Applicativo

Il contesto applicativo di questo progetto è quello della computer vision, settore fortemente influenzato negli ultimi anni dall'avvento delle Convolutional Neural Networks (CNN), un insieme di metodologie di elaborazione ispirate a processi biologici che avvengono nella corteccia visiva. Le CNN hanno recentemente consentito di ridurre significativamente il tasso di errore di classificazione, permettendo di migliorare l'accuratezza di sistemi di visione artificiale.

Le CNN sono tuttavia molto onerose dal punto di vista computazionale, e sono infatti spesso utilizzate su potenti GPU, la cui potenza assorbita si aggira spesso nell'ordine delle centinaia di Watt. Le GPU hanno inoltre lo svantaggio di rendere necessari sistemi di raffreddamento costosi in termini di spazio e consumi, ciò rende difficile il loro utilizzo in presenza di vincoli di spazio, peso o consumi. D'altro canto i sistemi embedded consentono di ottenere prestazioni confrontabili con quelle di moderne GPU, mantenendo limitati consumi (nell'ordine delle unità di Watt), dimensioni, peso e temperature (i sistemi di raffreddamento necessari sono spesso molto semplici e di dimensioni molto ridotte). Nonostante i numerosi vantaggi, lo svantaggio principale di tali sistemi è tuttavia quello della flessibilità, volendo modificare la programmazione è infatti necessario ripetere il processo di sintesi e implementazione del nuovo hardware, operazione che spesso richiede tempi di computazione non indifferenti.

Un ambito di ricerca di notevole interesse negli ultimi anni, la embedded computer vision, è mirato a implementare algoritmi e metodologie di visione artificiale (anche basate su reti neurali come CNN), su dispositivi embedded, rendendo fruibili tali funzionalità anche in presenza di stretti vincoli relativi alle risorse energetiche, ingombro e peso, consentendo di realizzare sistemi di elaborazione estremamente efficienti per esempio su droni, sistemi mobile e wearable.

Capitolo 2 - Strumenti Utilizzati

2.1 - Hardware

2.1.1 - ZedBoard

ZedBoard [4] è una scheda di development dai costi contenuti dotata di un AP SoC (All Programmable System on Chip) Xilinx Zynq 7000 [11]. La scheda è provvista di numerose interfacce e funzionalità di supporto per lo sviluppo di una grande varietà di applicazioni.

Il SoC Xilinx Zynq 7000, dotato di due processori ARM Cortex A9, integrati con una potente logica programmabile Artix-7 a 28nm, è un dispositivo che unisce la programmabilità software tipica di un processore ARM con la programmabilità hardware di un FPGA.

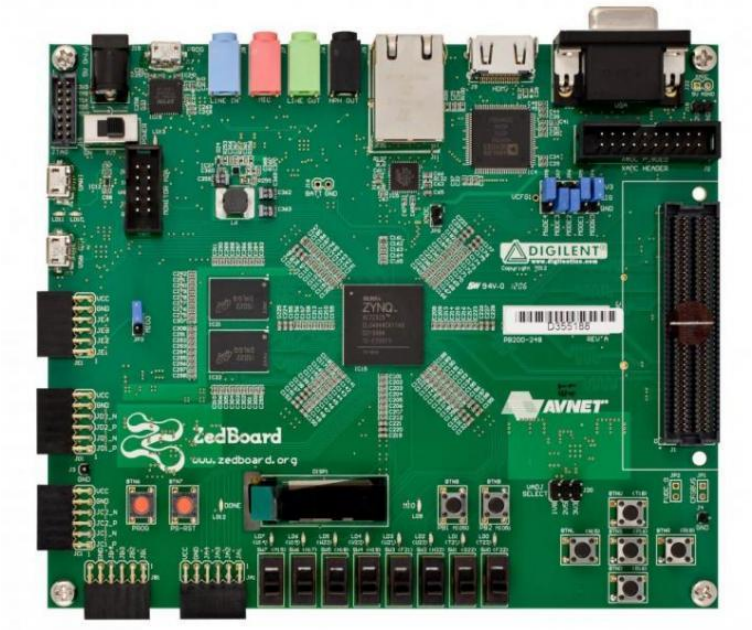


Figura 1: Avnet ZedBoard

Un dispositivo FPGA [7] (Field Programmable Gate Array) rende possibile la realizzazione di sistemi ad elevate prestazioni, pur mantenendo molto ridotti i consumi e di conseguenza le temperature. Le logiche programmabili FPGA sono adatte allo sviluppo di dispositivi logici ad elevata scala di integrazione e consentono un maggiore livello di astrazione rispetto alle tecnologie PAL, PLA e PLD.

I dispositivi FPGA sono usate tipicamente per la realizzazione di reti sequenziali sincrone e algoritmi. Sono quindi dotati di almeno un segnale di clock da inviare alle RSS che compongono il progetto. Spesso, aumentando la complessità del progetto, è necessario introdurre più domini di clock, vale a dire suddividere il progetto in diversi moduli che utilizzano ciascuno un clock diverso (in termini di frequenza, fase, e duty-cycle). Per generare segnali di clock stabili partendo da un segnale di clock esterno, modificando frequenza, fase e duty-cycle, e ridurre skew e jitter, i manufacturer di FPGA mettono tipicamente a disposizione componenti come Digital Clock Manager (DCM) e Phase Locked Loop (PLL).

I segnali di clock esterni hanno frequenze che si aggirano nell'ordine di grandezza delle centinaia di MHz, che appaiono ridotte se confrontate con quelle dei clock di alcune moderne CPU o GPU, che invece oscillano nell'ordine delle unità di GHz. Per ottenere prestazioni elevate si sfrutta quindi, oltre a un elevato e configurabile grado di parallelismo, la possibilità offerta dalla programmabilità hardware di eseguire in un singolo ciclo di clock una quantità di operazioni che una CPU eseguirebbe in decine o centinaia.

Un FPGA è costituito da una matrice di migliaia blocchi logici configurabili (CLB), memorie RAM a blocchi (BRAM), e alcune decine di sommatori e moltiplicatori. Ogni CLB è composto da un insieme di slice, ognuna contenente una serie di Logic Cell (LC), che costituiscono l'unità di base di un dispositivo FPGA.

Ogni LC è composta da una Look Up Table (LUT) programmabile, un Flip Flop D (FFD) per il campionamento del risultato della LUT e un multiplexer che manda in uscita o il risultato della LUT, o il valore campionato all'ultimo fronte di salita del clock.

La programmabilità di tali CLB consente di realizzare qualsivoglia applicazione, anche ad elevata scala di integrazione, e, tramite alcuni strumenti di sintesi, di mantenere un elevato livello di astrazione, consentendo di sintetizzare blocchi hardware utilizzando linguaggi di alto livello come C o C++.

2.1.2 - Sensori d'immagine OV7670

Omnivision OV7670 [5] è un sensore d'immagine a basso voltaggio, dai costi ridotti, costituito da una singola camera capace di produrre uno stream di immagini 640x480 a 30 fotogrammi al secondo.

OV7670 supporta vari formati di output a 8 bit:

- YUV/YCbCr 4:2:2
- RGB 565/555
- GRB 4:2:2
- Raw RGB Data

Nel caso trattato, il sensore trasmette le immagini acquisite in spazio di colore YUV alla ZedBoard e viene configurato utilizzando il protocollo I2C.

Figura 2: Sensore d'immagine OV7670

2.2 - Ambienti di Sviluppo

L'ambiente di sviluppo scelto per la realizzazione di questo progetto è Xilinx Vivado [6] (versione 2016.2), di cui sono stati utilizzati principalmente i tre seguenti moduli:

- Vivado IP Integrator
- Vivado HLS
- Vivado SDK

Per la realizzazione del client è stato utilizzato CodeBlocks [14], un versatile IDE multi-piattaforma, ideale per lo sviluppo in C e C++.

2.2.1 - Vivado IP Integrator

Il primo modulo utilizzato, Vivado IP Integrator è un potente strumento di sintesi e validazione di design a blocchi denominati IP cores. Un IP core (Intellectual Property Core) è rappresentato da un blocco che implementa una determinata rete logica o algoritmo, utilizzata nella sintesi di design hardware, nel caso specifico di questa tesi, nel contesto applicativo della programmazione di un FPGA. L'utilizzo di IP cores consente un alto livello di astrazione, e di riusabilità del design.

Lo strumento principale di Vivado IP Integrator è una interfaccia drag and drop dove vengono inseriti, connessi e configurati i suddetti IP cores, rappresentati da blocchi rettangolari.

Ogni IP core possiede un certo numero di ingressi e uscite (a cui è associato un nome e un tipo di interfaccia). Ogni uscita può essere connessa ad un numero arbitrario di ingressi, a ogni ingresso può essere collegata una sola uscita. Per garantire l'alto livello di astrazione, un insieme di IP core collegati e configurati in un determinato modo può essere raggruppato in una hierarchy, che viene trattata in modo uniforme agli IP cores, e può essere inserita a sua volta in un'altra hierarchy di più alto livello. La riusabilità è invece garantita dalla possibilità di configurare ogni IP-core separatamente, e ciò rende possibile la sintesi di blocchi dal comportamento generico e dipendente da una configurazione che non richiede la modifica del sorgente (e la conseguente sintesi).

Vivado IP Integrator offre inoltre la possibilità di validare un determinato design hardware, e di generare il corrispondente file di bitstream (.bit) per la programmazione della FPGA

in modo trasparente al programmatore e quasi completamente automatico. L'intervento del programmatore è richiesto solo nel caso in cui si vogliano apportare modifiche specifiche al design sintetizzato.

È inoltre possibile effettuare test e debug del design sintetizzato mediante strumenti di analisi e simulazione.

2.2.2 - Vivado HLS

Il secondo modulo utilizzato è Vivado HLS, un IDE basato su Eclipse, finalizzato alla sintesi degli IP cores introdotti nel paragrafo precedente. Questo strumento consente di sintetizzare IP cores utilizzabili in Vivado IP Integrator partendo da linguaggi di più alto livello come C e C++. Nel caso trattato in questa tesi, il riferimento per la sintesi di IP cores è il linguaggio C.

Il codice C (o C++) viene compilato in Verilog o VHDL (a scelta), per poi essere compilato nuovamente in RTL, e inserito in una repository accessibile da Vivado IP Integrator. Il processo di sintesi di alto livello è capace di sintetizzare IP cores dalle prestazioni comparabili a quelle ottenibili sintetizzando direttamente a partire dal codice RTL.

Ogni IP core viene implementato come una funzione C (top function), con determinati parametri in ingresso e in uscita. È naturalmente possibile distribuire il codice su diverse funzioni invocate dalla top function e su diversi moduli. Tramite un insieme di direttive è possibile associare ad ogni porta in ingresso e uscita, un'interfaccia e la relativa configurazione. Le direttive influiscono inoltre sul comportamento di HLS durante la sintesi, per introdurre ottimizzazioni, o comportamenti particolari diversi da quello predefinito.

Vivado HLS mette a disposizione del programmatore strumenti di validazione e simulazione, consentendo di effettuare test del funzionamento del singolo IP core, senza rendere necessaria la loro integrazione. La simulazione offre anche la possibilità di conoscere a priori delle stime temporali della latenza, throughput, intervalli, e una tabella che mostra in dettaglio l'utilizzo delle risorse della FPGA da programmare.

2.2.3 - Vivado SDK

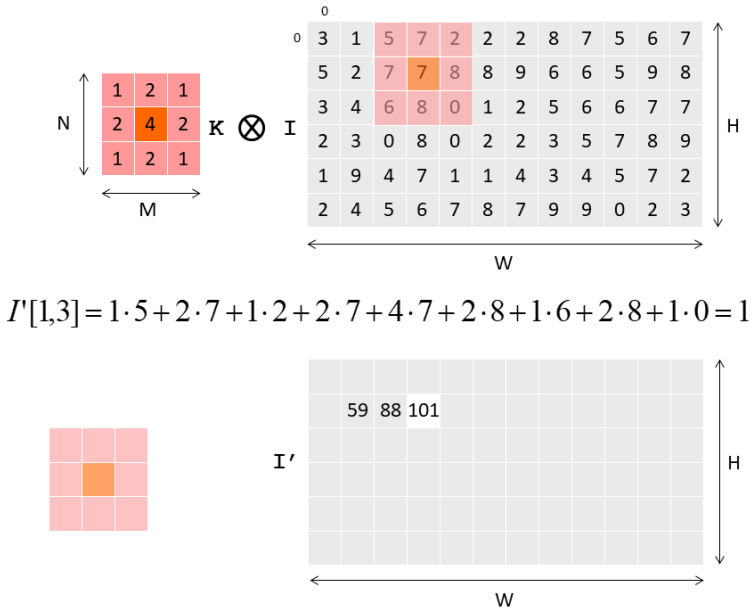
Il terzo modulo utilizzato è Vivado SDK, un altro IDE basato su Eclispe, questo finalizzato invece allo sviluppo di applicazioni software che dovranno eseguire sull'hardware prece-

dentemente sintetizzato. Vivado SDK mette a disposizione strumenti di debug, simulazione e supporto; consente di visualizzare l'output ricevuto tramite JTAG UART del System Debugger su un terminale.

Capitolo 3 - Filtro di Convoluzione

Il processo di filtraggio delle immagini avviene mediante l'operazione di convoluzione [7] tra due matrici bidimensionali: la prima matrice è l'immagine da filtrare, la seconda viene definita kernel e i suoi valori, "pesi". In genere il kernel consiste di una matrice di piccole dimensioni rispetto all'immagine da filtrare, nel caso di questo progetto, le dimensioni del kernel sono limitate superiormente a 7.

Il kernel di dimensioni N*M viene fatto "scorrere" sull'immagine in ingresso, ogni pixel dell'immagine in uscita è il risultato della media pesata dei pixel dell'immagine in ingresso che costituiscono un rettangolo N*M centrato sul pixel da determinare. Questo significa che utilizzando un kernel N*M, per ogni pixel devono essere effettuate N*M prodotti e N*M – 1 somme. Cambiando i pesi del kernel è possibile ottenere diversi tipi di filtro, riutilizzando lo stesso algoritmo per ottenere operazioni di filtering diverse (figura 3 e 4).



$$I'[1,3] = 1\cdot 5 + 2\cdot 7 + 1\cdot 2 + 2\cdot 7 + 4\cdot 7 + 2\cdot 8 + 1\cdot 6 + 2\cdot 8 + 1\cdot 0 = 101$$

Figura 3: Esempio di calcolo di un pixel mediante convoluzione

Sfruttando l'accelerazione hardware fornita dalla logica programmabile, si è scelto di implementare il filtro di convoluzione come un ip core, sintetizzandolo tramite Vivado HLS.

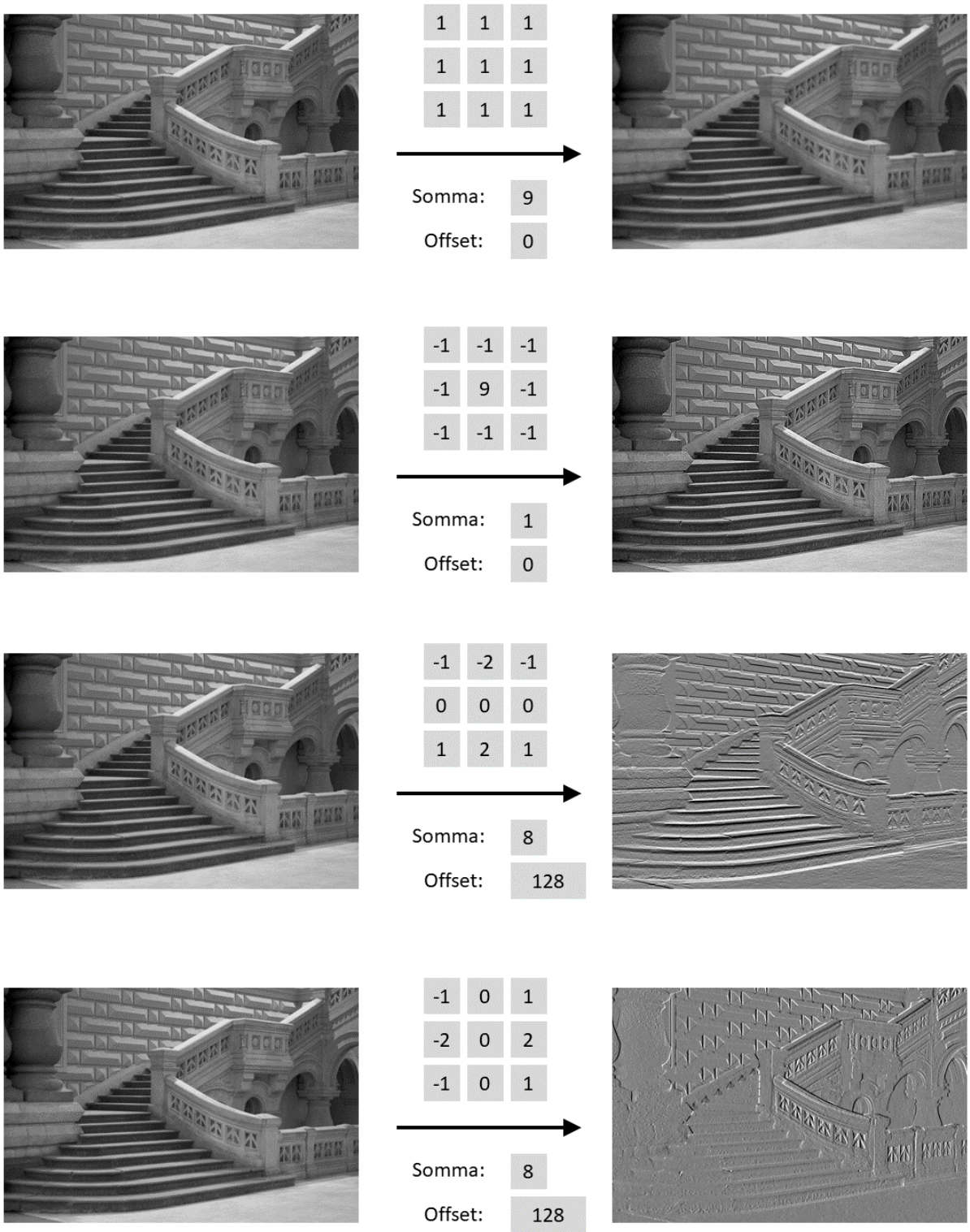


Figura 4: Esempio di alcune operazioni di filtering mediante convoluzione. Nell'ordine: mean, sharpen, sobel h, sobel v.

Si è deciso di rendere indipendente dal kernel il valore della somma dei pesi, liberamente impostabile insieme a un parametro “offset” che viene aggiunto al valore di ogni pixel al termine dell’elaborazione. Questo è necessario per evitare che in alcune elaborazioni, alcuni pixel non risultino corretti in seguito al troncamento dei valori in 8 bit. Se dopo un’elaborazione un pixel assume un valore fuori dal range [0..255], questo verrà interpretato come byte senza segno, producendo un effetto simile a quello mostrato in figura.

Figura 5: Esempio di artefatti in presenza di pixel fuori range

Alternativamente, il problema potrebbe essere gestito inserendo dei margini di saturazione, in modo che tutti i pixel negativi vengano settati a 0, e tutti quelli oltre il valore 255, vengano settati a 255. Così facendo sarebbero però necessari dei controlli ad ogni calcolo, diminuendo l’efficienza e aumentando le risorse necessarie.

Il numero di pixel significativi dell’immagine elaborata è inferiore rispetto a quello dell’immagine in ingresso, infatti non risultano significativi i pixel vicini ai bordi dell’immagine (precisamente tutti i pixel che distano al massimo KERNEL_MID_WIDTH da uno dei due bordi verticali o KERNEL_MID_HEIGHT da quelli orizzontali), questo perché in quelle aree, parte della processing window risiede fuori dall’immagine.

Non gestendo il problema, ad ogni convoluzione, l’immagine risulterebbe di dimensioni inferiori, in quanto i pixel ai bordi non sono calcolati. Alternativamente è possibile introdurre dei pixel artificiali (padding), per mantenere le dimensioni dell’immagine costanti. Per risparmiare sull’utilizzo di LUT e MUX, si è scelto di non distinguere tra pixel di padding e pixel significativi dell’immagine, lasciando non correttamente inizializzati i registri della processing window e calcolando indifferentemente il risultato.

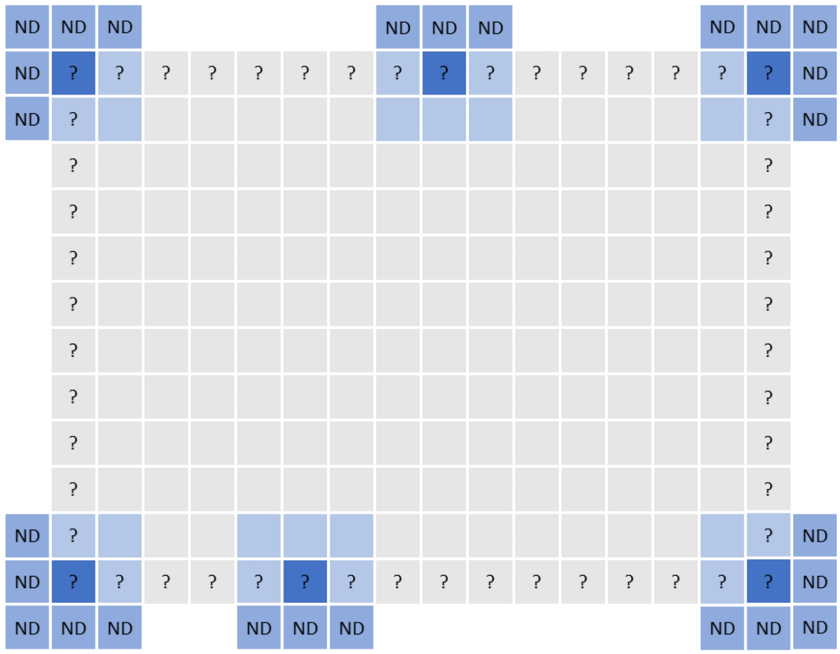


Figura 6: Illustrazione del problema degli effetti ai bordi

Nella seguente illustrazione viene mostrato l'effetto del padding ai bordi delle immagini elaborate. L'immagine è stata elaborata utilizzando un kernel 7x7, il padding consiste di un bordo spesso 3 pixel, la metà delle dimensioni del kernel.

Figura 7: Padding ai bordi di un'immagine elaborata

3.1 - File configurable_convolution_filter.h

```
#ifndef CONFIGURABLE_CONVOLUTION_FILTER
#define CONFIGURABLE_CONVOLUTION_FILTER

#include "ap_int.h"

#define BIT_ACCURATE

#define IMG_HEIGHT 480
#define IMG_WIDTH 640

#define KERNEL_HEIGHT 7
#define KERNEL_WIDTH 7

//Default 8
#define PIXEL_BITS 8
//Default 8
#define WEIGHT_BITS 8
//Set this to at least
//ceil(log_2(PIXEL_BITS * WEIGHT_BITS * KERNEL_HEIGHT * KERNEL_WIDTH))
#define MAC_BITS 22
//Set this to at least PIXEL_BITS + WEIGHT_BITS
#define MUL_BITS 16

typedef ap_uint<PIXEL_BITS> pixel;
typedef ap_int<WEIGHT_BITS> s_int;

#define END_PARAMS 2

const int CONFIG_LEN = KERNEL_HEIGHT * KERNEL_WIDTH + END_PARAMS;
const int KERNEL_SUM_INDEX = CONFIG_LEN - 2;
const int KERNEL_OFF_INDEX = CONFIG_LEN - 1;

const int KERNEL_MID_HEIGHT = (KERNEL_HEIGHT - 1) / 2;
const int KERNEL_MID_WIDTH = (KERNEL_WIDTH - 1) / 2;

pixel pixel_weighted_average(s_int kernel[KERNEL_HEIGHT][KERNEL_WIDTH],
              s_int kern_sum,
              s_int kern_off,
              pixel window[KERNEL_HEIGHT][KERNEL_WIDTH]);

void convolution_filter(s_int kernel_config[CONFIG_LEN],
              pixel in_img[IMG_HEIGHT*IMG_WIDTH],
              pixel out_img[IMG_HEIGHT*IMG_WIDTH]);

#endif
```

Il file “configurable_convolution_filter.h”, contiene le definizioni delle costanti utilizzate:

- IMG_HEIGHT e IMG_WIDTH: rappresentano l’altezza e la larghezza in pixel dell’immagine in ingresso e uscita.
- KERNEL_HEIGHT e KERNEL_WIDTH: rappresentano le dimensioni della matrice kernel.
- PIXEL_BITS: indica il numero di bit dei registri che memorizzano i pixel dell’immagine.

- WEIGHT_BITS: indica il numero di bit dei registri che memorizzano i pesi del kernel e i parametri di configurazione aggiuntivi.
- MAC_BITS: indica il numero di bit del registro utilizzato per memorizzare il valore temporaneo della somma nelle operazioni MAC.
- MUL_BITS: indica il numero di bit del registro utilizzato per memorizzare il valore temporaneo del prodotto nelle operazioni MAC.
- END_PARAMS: indica il numero di parametri aggiuntivi presi in ingresso oltre ai pesi del kernel.

Vengono ridefiniti due tipi dati:

- "s_int" rappresenta un valore intero con segno, a WEIGHTS_BITS numero di bit.
- "pixel" rappresenta un valore intero senza segno, utilizzato esclusivamente per i pixel dell'immagine, a PIXEL_BITS numero di bit.

La funzione "convolution_filter" costituisce la top function dell'ip core, essa prende in ingresso:

- Un array di "s_int" contenente i pesi del kernel, e i parametri aggiuntivi.
- Un array di "pixel" contenente l'immagine in ingresso.
- Un array di "pixel" utilizzato come uscita per l'immagine elaborata.

La funzione "pixel_weighted_average" prende in ingresso:

- Una matrice di "s_int" che rappresenta il kernel
- Un "s_int" contenente il valore della somma.
- Un "s_int" contenente il valore dell'offset.
- Una matrice di "pixel" che rappresenta una porzione dell'immagine avente le stesse dimensioni del kernel (processing window).

La funzione restituisce un pixel contenente il risultato dell'elaborazione, ovvero la media pesata dei pixel della processing window, a cui viene aggiunto il valore dell'offset.

3.2 - File configurable_convolution_filter.c

3.2.1 - Porte e interfacce

Il file "configurable_convolution_filter.c" definisce le funzioni dichiarate in "configurable_convolution_filter.h". Per ogni parametro della funzione "convolution_filter", viene inserita una direttiva che via associa un'interfaccia.

Si è scelto di utilizzare:

- AXI_STREAM per l'immagine in ingresso e in uscita, questo consente di inviare e ricevere dati su canali di comunicazione uno a uno con altri ip core.
- AXI_LITE per i pesi del kernel e i parametri aggiuntivi, questo consente di effettuare operazioni di scrittura da software, dopo aver mappato questo ip core nello spazio di indirizzamento del PS.

Le due porte in ingresso mettono a disposizione un nuovo byte a ogni ciclo del clock, similmente, a ogni ciclo il filtro produrrà in uscita un pixel (un byte).

Vengono di seguito mostrate le direttive associate a ciascuna porta, la direttiva commentata, "HLS RESOURCE" [8], servirebbe per limitare l'utilizzo di BRAM da parte dell'interfaccia axilite nella porta "filter", è stata commentata a causa di un bug [10] presente nella versione di Vivado utilizzata per questo progetto, che ne impedisce l'utilizzo.

```
void convolution_filter(s_int kernel_config[CONFIG_LEN],
              pixel in_img[IMG_HEIGHT*IMG_WIDTH],
              pixel out_img[IMG_HEIGHT*IMG_WIDTH])
{
//#pragma HLS RESOURCE variable=kernel_config core=RAM_S2P_LUTRAM
#pragma HLS INTERFACE s_axilite port=kernel_config
#pragma HLS INTERFACE axis port=out_img
#pragma HLS INTERFACE axis port=in_img

      //...

}
```

3.2.2 - Strutture dati utilizzate

Il filtro di convoluzione fa uso di tre strutture dati di bufferizzazione:

- "line_buffer": una matrice di pixel, avente larghezza pari alla larghezza dell'immagine, e altezza inferiore di un'unità rispetto a quella del kernel.
- "window": una matrice di pixel, avente le stesse dimensioni del kernel.
- "kernel": una matrice di byte con segno, contenente i pesi del kernel.

```
        //line buffer
       static pixel line_buffer[KERNEL_HEIGHT - 1][IMG_WIDTH];
#pragma HLS ARRAY_PARTITION variable=line_buffer complete dim=1

       //processing window
       static pixel window[KERNEL_HEIGHT][KERNEL_WIDTH];
#pragma HLS ARRAY_PARTITION variable=window complete dim=0

       //kernel_config
       static s_int kernel[KERNEL_HEIGHT][KERNEL_WIDTH];
#pragma HLS ARRAY_PARTITION variable=kernel complete dim=0
```

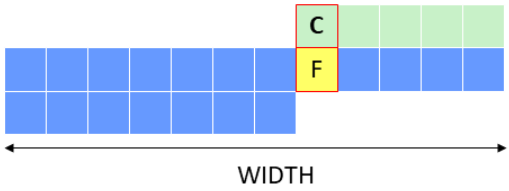


Figura 8: Illustrazione array "line_buffer"

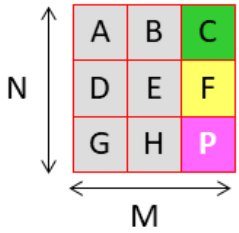


Figura 9: Illustrazione array "window"

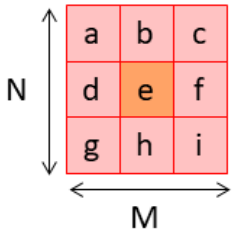


Figura 10: Illustrazione array "kernel"

Utilizzando la direttiva "HLS ARRAY_PARTITION" [8] è possibile partizionare array bidimensionali in una serie di array lineari, o direttamente in singoli registri.

Oltre ai buffer precedentemente introdotti, vengono memorizzati i valori che descrivono la somma del kernel e l'offset in due registri "kern_sum" e "kern_off". I due registri "i" e "j" serviranno durante la fase di inizializzazione, consentendo di evitare l'utilizzo di un DSP. Il funzionamento verrà descritto nel prossimo paragrafo.

```
       static s_int kern_sum = 1;
       static s_int kern_off = 0;

       static int i = 0;
       static int j = 0;

       i = 0;
       j = 0;
```

3.2.3 - Pipeline di esecuzione

L'algoritmo è composto da due cicli for innestati, che iterano orizzontalmente e verticalmente, su un numero di righe pari a IMG_HEIGHT + KERNEL_MID_HEIGHT (480 + 3) e un numero di colonne pari a IMG_WIDTH + KERNEL_MID_WIDTH (640 + 3).

```
Loop_row: for(int row = 0; row < IMG_HEIGHT + KERNEL_MID_HEIGHT; row++)
              Loop_col: for(int col = 0; col < IMG_WIDTH + KERNEL_MID_WIDTH; col++)
              {
#pragma HLS PIPELINE II=1

                    //...

              }
```

Questi cicli annidati subiscono una operazione di "flattening" ovvero vengono trasformati in un unico loop senza loop annidati, e la sua esecuzione avviene in pipeline (utilizzando la direttiva "HLS PIPELINE" [8], consentendo di migliorare il throughput della rete.

Supponendo che siano necessari N cicli di clock per completare un'operazione, senza parallelismo, il completamento di M operazioni avviene dopo N*M cicli di clock. Parallelizzando mediante pipeline, l'operazione viene idealmente suddivisa in N step, ognuno eseguito in un ciclo di clock. Mentre l'operazione A si trova nello step K, l'operazione successiva B si trova nello step K – 1, C si trova in K – 2, eccetera, consentendo di realizzare in parallelo idealmente al massimo N operazioni, riducendo la latenza totale a da N*M a N + M – 1 cicli di clock.

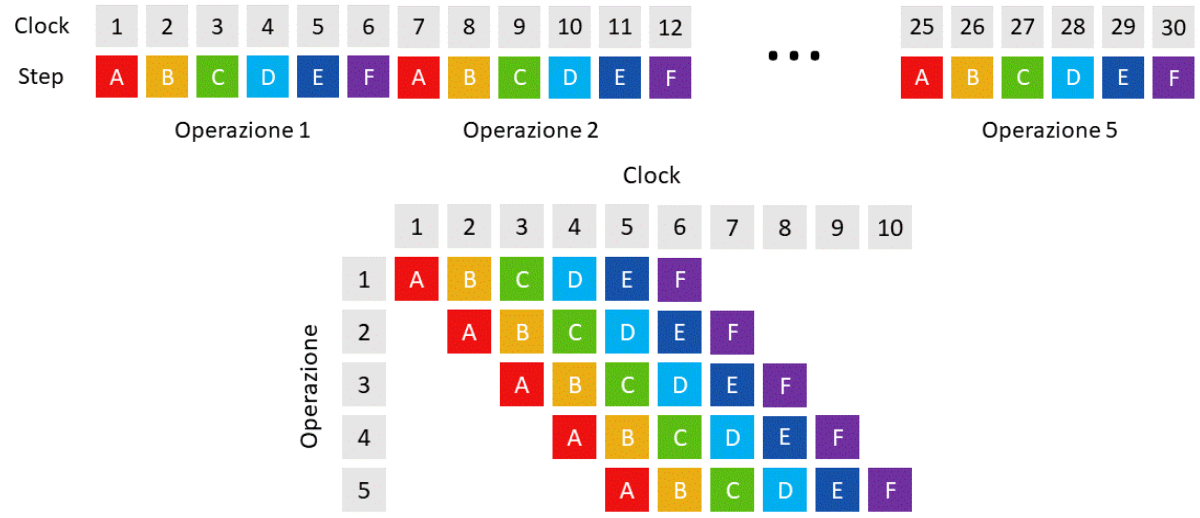


Figura 11: Confronto tra esecuzione sequenziale e in pipeline

Ad ogni iterazione viene determinato il valore della variabile "iteration", che indica il numero dell'iterazione corrente. Questo valore viene utilizzato in fase di configurazione

del filtro che avviene infatti nelle prime CONFIG_LEN (7 * 7 + 2 = 51) iterazioni. Ad ogni iterazione l'interfaccia axilite mette a disposizione un nuovo valore, che deve essere inserito nella posizione corretta della matrice "kernel", indicata dai registri precedentemente introdotti: "i" e "j", che verranno conseguentemente aggiornati. Gli ultimi due valori, corrispondenti agli indici 49 e 50, vengono memorizzati nei registri "kern_sum" e "kern_off". La configurazione del kernel viene effettuata una volta per ogni elaborazione.

```
const int iteration = row * IMG_WIDTH + col;
//kernel_config setup

if (iteration < KERNEL_HEIGHT * KERNEL_WIDTH)
{
        if (j >= KERNEL_WIDTH)
        {
                j = 0;
                i++;
        }
        kernel[i][j] = kernel_config[iteration];
        j++;
}
else if (iteration == KERNEL_SUM_INDEX)
        kern_sum = kernel_config[KERNEL_SUM_INDEX];
else if (iteration == KERNEL_OFF_INDEX)
        kern_off = kernel_config[KERNEL_OFF_INDEX];
```

Alla fase di configurazione (che viene eseguita solo una volta per elaborazione), segue lo shift delle colonne della processing window. La processing window infatti deve scorrere sull'immagine, da sinistra a destra, dall'alto in basso, vengono quindi copiati i valori della colonna "j + 1" nella colonna "j" per le prime KERN_WIDTH – 1 colonne.

```
//shift processing window columns
for(int i = 0; i < KERNEL_HEIGHT; i++)
        for(int j = 0; j < KERNEL_WIDTH - 1; j++)
                window[i][j] = window[i][j+1];
```

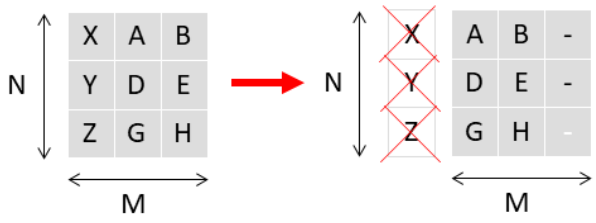


Figura 12: Shift delle colonne della processing window
(l'esempio illustra per semplicità un kernel 3x3)

Si procede quindi a popolare l'ultima colonna della processing window, si utilizza a tale scopo il line buffer, che mantiene le ultime KERNEL_HEIGHT – 1 righe dell'immagine in ingresso (il popolamento del line buffer verrà illustrato nel paragrafo seguente). Subito dopo la copia del pixel dal line buffer alla processing window, viene shiftata in alto la colonna occupata dai pixel copiati del line buffer, liberando lo spazio per i nuovi pixel.

```
//copy KERN_H - 1 values from line_buffer to processing window
                if(col < IMG_WIDTH)
                {
                        for(int ii = 0; ii < KERNEL_HEIGHT - 2; ii++)
                        {
                                window[ii][KERNEL_WIDTH - 1] = line_buffer[ii][col];
                                line_buffer[ii][col] = line_buffer[ii + 1][col];
                        }
                        line_buffer[KERNEL_HEIGHT - 2][col] =
                                        line_buffer[KERNEL_HEIGHT - 1][col];


                }
```

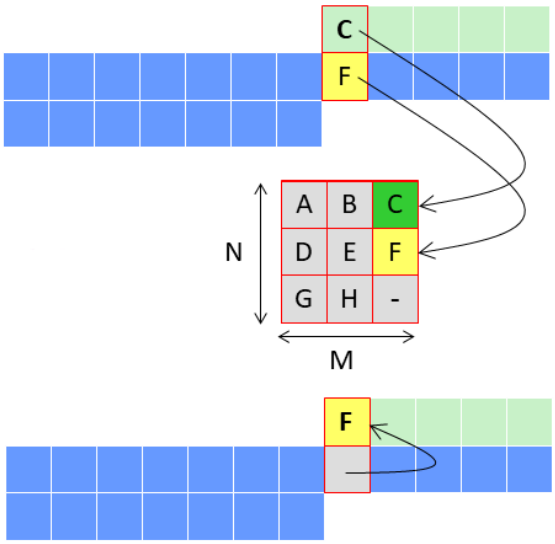


Figura 13: Popolamento dell'ultima colonna della processing window e shift verso l'alto del line buffer

L'ultimo pixel della processing window viene letto dallo stream in input, e inserito contemporaneamente sia nella processing window che nel line buffer, completando l'aggiornamento dei buffer.

Terminata la fase di aggiornamento dei buffer, nel caso in cui la processing window contenga una porzione di immagine contigua (si scartano le prime KERN_MID_HEIGHT righe e le prime KERN_MID_WIDTH colonne), verrà calcolato un pixel dell'immagine in uscita, e scritto sullo stream in output.

```
//input value
if(col < IMG_WIDTH && row < IMG_HEIGHT)
{
        pixel in_temp = in_img[row * IMG_WIDTH + col];
        window[KERNEL_HEIGHT - 1][KERNEL_WIDTH - 1] = in_temp;
        line_buffer[KERNEL_HEIGHT - 2][col] = in_temp;
}

//output value
if (row >= KERNEL_MID_HEIGHT && col >= KERNEL_MID_WIDTH)
{
        pixel out = pixel_weighted_average(
                        kernel, kern_sum, kern_off, window);
        out_img[(row - KERNEL_MID_HEIGHT) * IMG_WIDTH
                        + (col - KERNEL_MID_WIDTH)] = out;
}
```

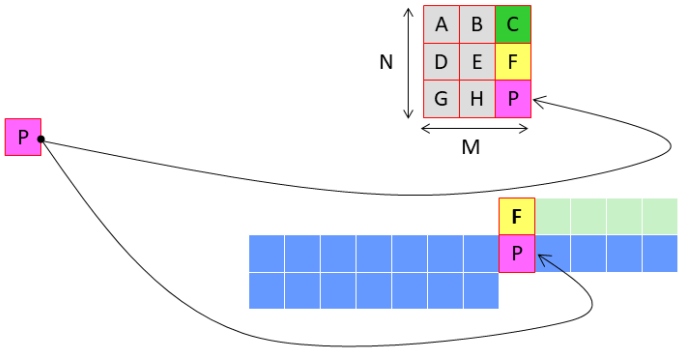


Figura 14: Completamento della bufferizzazione

Come è osservabile, la funzione che calcola il pixel in uscita viene invocata la prima volta alla riga KERNEL_MID_HEIGHT e alla colonna KERNEL_MID_WIDTH, ovvero alla iterazione numero 1932. È pertanto impossibile che il calcolo del pixel in uscita avvenga senza aver prima configurato il kernel, essendo necessarie solo 51 iterazioni per farlo.

Nel dettaglio, il calcolo del pixel in uscita avviene nella funzione “pixel_weighted_average”, che calcola il prodotto scalare tra le due matrici prese in ingresso, divide il risultato per la somma specificata, e aggiunge infine al risultato il valore specificato come offset.

Tra le direttive utilizzate vi è “HLS INLINE” [8] che consente di sintetizzare la funzione chiamata come un’entità unica con la funzione chiamante, consentendo di condividere le stesse ottimizzazioni applicate alle operazioni circostanti. La direttiva HLS RESOURCE influisce sulle risorse utilizzate in fase di sintesi, verrà discussa nel paragrafo di valutazione delle performance.

```
pixel pixel_weighted_average(s_int kernel[KERNEL_HEIGHT][KERNEL_WIDTH],
               s_int kern_sum,
               s_int kern_off,
               pixel window[KERNEL_HEIGHT][KERNEL_WIDTH])
{
#pragma HLS INLINE

       ap_int<MAC_BITS> out_temp = 0;
       ap_int<MUL_BITS> temp = 0;
#pragma HLS RESOURCE variable=temp core=Mul_LUT

       Edge_i:  for(int i = 0; i < KERNEL_HEIGHT; i++)
               Edge_j: for(int j = 0; j < KERNEL_WIDTH; j++){
                       temp = window[i][j] * kernel[i][j];
                       out_temp = out_temp + temp;
               }

       return ((out_temp / kern_sum) + kern_off)(PIXEL_BITS - 1,0);
}
```

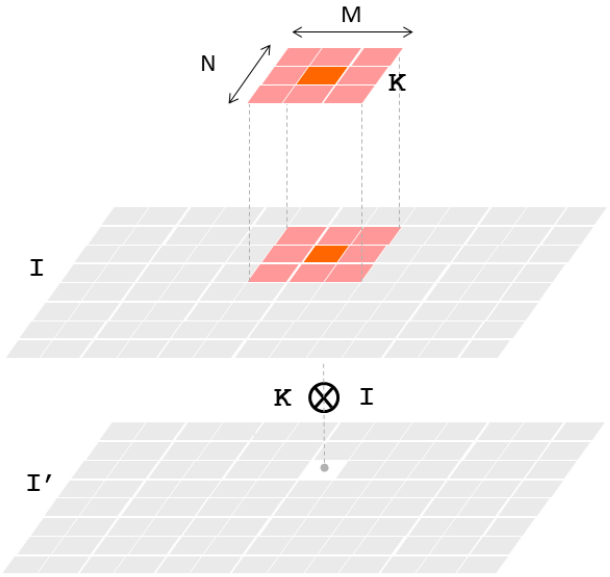


Figura 15: Calcolo di un pixel dell'immagine in uscita

3.3 - Considerazioni sull'estendibilità

Nel file "configurable_convolution_filter.h" compaiono alcune costanti cablate, ciò produce alcuni vantaggi e svantaggi.

I vantaggi consistono in:

- Semplicità del codice.
- Ottimizzazione automatica da parte di Vivado HLS.
- Performance migliori.
- Tempi di sintesi ridotti.

Gli svantaggi d'altro canto sono:

- Configurabilità limitata.
- Necessità di intervenire modificando il sorgente per cambiare il comportamento dell'ip core.

Si è tuttavia preferita questa realizzazione perché le responsabilità di questo ip core sono ben definite e difficilmente variabili: gli unici punti di estensione potrebbero essere l'aggiunta di eventuali altri parametri allo stream in ingresso, mentre per aumentare le dimensioni del kernel o dell'immagine, basta cambiare il valore di alcune costanti. Introdurre la possibilità di inserimento di kernel di dimensione variabile non porterebbe alcun vantaggio nell'utilizzo di risorse sull'FPGA. Queste infatti sono programmate staticamente prima dell'avvio dell'applicazione. Anzi, sarebbero necessarie più risorse per gestire la maggiore complessità logica e gli eventuali casi d'eccezione, senza considerare l'overhead che questa dinamicità comporterebbe. La sintesi della pipeline risulterebbe infatti più complessa in seguito all'introduzione di cicli con un numero di iterazioni variabile e l'accesso non sequenziale e non deterministico ai registri.

3.4 - Integrazione tramite Vivado IP Integrator

Per integrare il nuovo filtro di convoluzione all'interno del diagramma a blocchi è necessario rimuovere la possibilità di impostare il filtro tramite gli switch collocati sulla board. La selezione del filtro avveniva infatti tramite "id_filter" ovvero un valore di 3 bit che consentiva di scegliere uno tra 8 filtri preimpostati; questa funzionalità è stata rimossa e sostituita da una porta axilite memory-mapped che consente di configurare il filtro dal software in esecuzione sul processore ARM.

Per integrare il nuovo filtro di convoluzione, sarebbe sufficiente creare una nuova uscita nel blocco AXI Interconnect, collegarla alla porta di configurazione del filtro, e effettuare il mapping del nuovo dispositivo nello spazio di indirizzamento del PS. Tuttavia, per facilitare ed uniformare il processo di debug e testing si è preferito collegare in serie il filtro precedentemente descritto con una versione separabile [9] sviluppata da Marco Rossini [3]. In questo modo è possibile eseguire test su entrambi i filtri separatamente, utilizzando lo stesso diagramma a blocchi, senza necessità di re-sintetizzare. Nonostante il collegamento in serie, è comunque possibile verificare separatamente il comportamento di ciascuno dei due filtri, configurando l'altro come un filtro identità.

Per astrarre dal comportamento del filtro, si è deciso di sostituire l'ip_core del filtro non configurabile con una hierarchy "Convolution_Filter", mostrata nella figura sottostante (nella figura alcuni collegamenti sono stati omessi, per rendere più comprensibili le modifiche effettuate).

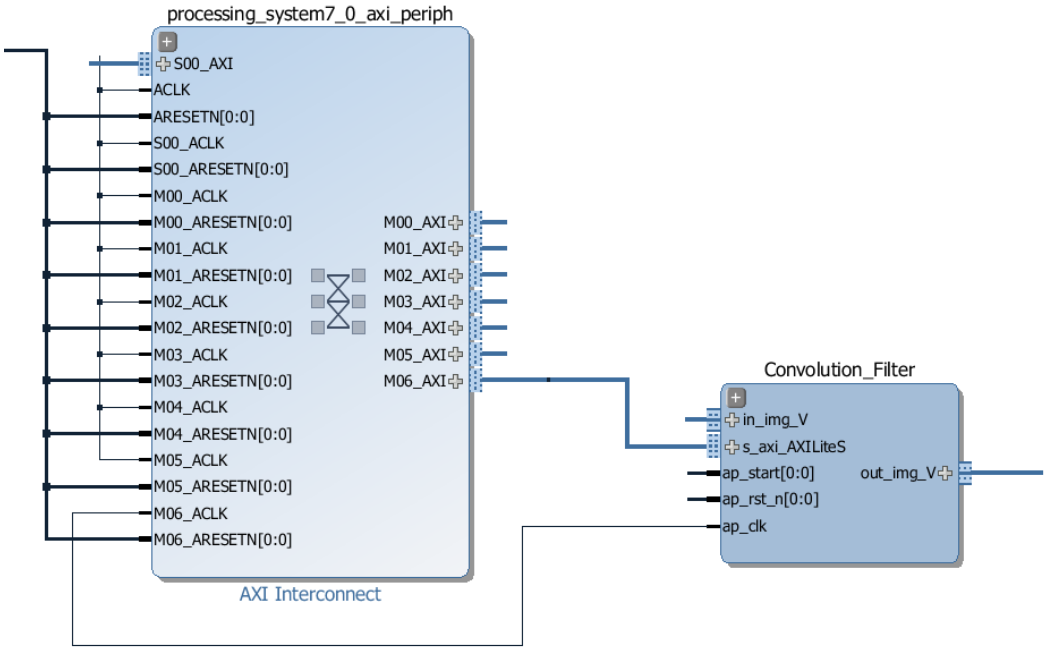


Figura 17: Integrazione del filtro di convoluzione configurabile

All'interno della hierarchy "Convolution_Filter" è presente un secondo blocco axi interconnect, a cui sono connessi come slave i due filtri. L'immagine in grayscale viene elaborata dal primo filtro "sep_convolution_filter_0" (discusso nella tesi di Marco Rossini) viene passata all'ingresso del secondo filtro "convolution_filter_0" (la cui realizzazione è stata discussa precedentemente), che produce in uscita l'immagine elaborata.

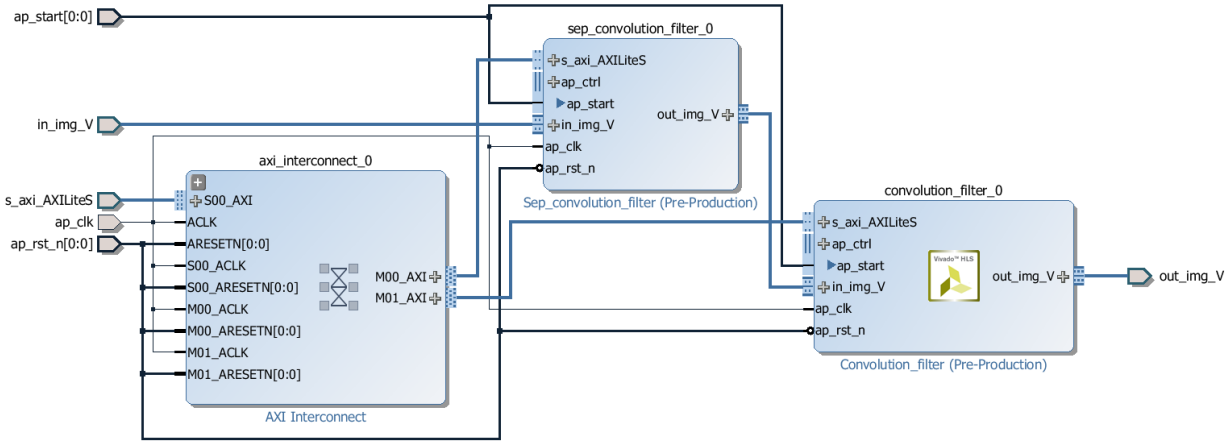


Figura 16: Contenuto della Hierarchy "Convolution_Filter"

Capitolo 4 - Software lato servitore

Le modifiche al software applicativo consistono nell'introduzione di un nuovo modulo che si occupa del controllo e configurazione del filtro di convoluzione, e l'estensione del modulo di rete preesistente dedicato alla connessione TCP, scritto da Mattia Bernasconi [1].

Il modulo di controllo del filtro di convoluzione si compone dei file "convolution_filter.h" e "convolution_filter.c", il modulo di rete si compone di "tcp_connection.h" e "tcp_connection.c".

4.1 - Modulo convolution_filter

Il modulo "convolution_filter" espone funzionalità di alto livello per il controllo e configurazione del filtro di convoluzione, fungendo da wrapper dei driver generati da Vivado HLS.

Viene definita la struttura dati "kernel_config" che mantiene le seguenti informazioni:

- Dimensioni del kernel (2 unsigned char)
- Pesi del kernel (un array di double)
- Somma del kernel (signed char)
- Offset (signed char)
- Numero di bit di precisione (unsigned char)

```
typedef struct kernel_config {
	unsigned char height;
	unsigned char width;
	double *weights;
	signed char sum;
	signed char offset;
	unsigned char bit_shift;
} kernel_config;
```

Vengono dichiarati due puntatori a kernel_config globali, uno che punta alla configurazione corrente ("kernel"), l'altro a una configurazione temporanea ("temp_config").

La funzione "convolution_filter_get_kernel_config" copia la configurazione corrente nell'area di memoria puntata da "temp_config", e restituisce "temp_config", in modo che eventuali modifiche affliggano solo l'area copiata. La funzione di utilità "copy_kernel_config" non viene esposta nell'header, e si occupa di clonare l'area di memoria puntata da "source" in "dest", procedendo ad eventuali allocazioni e de-allocazioni se necessario.

```
kernel_config* convolution_filter_get_kernel_config(void)
{
	copy_kernel_config(kernel, &temp_config);
	return temp_config;
}
```

```
void copy_kernel_config(kernel_config* source, kernel_config** dest)
{
	int i;

	if (*dest != NULL)
	{
		if ((*dest)->weights != NULL && (*dest)->weights != source->weights)
			free((*dest)->weights);

		free(*dest);
	}

	*dest = (kernel_config*)malloc(sizeof(kernel_config));

	(*dest)->bit_shift = source->bit_shift;
	(*dest)->height = source->height;
	(*dest)->width = source->width;
	(*dest)->sum = source->sum;
	(*dest)->offset = source->offset;
	(*dest)->weights = (double*)malloc(sizeof(double) * source->height * source->width);

	for (i = 0; i < (*dest)->height * (*dest)->width; i++)
	{
		(*dest)->weights[i] = source->weights[i];
	}
}
```

Viceversa, la funzione “convolution_filter_configure” riceve in ingresso un puntatore a “kernel_config”, effettua controlli di validità, nel caso in cui la nuova configurazione sia valida, viene generato uno stream di byte da scrivere via axilite all’indirizzo a cui è mappato il filtro, il filtro viene configurato, e nel caso in cui la configurazione avvenga correttamente, verrà aggiornato il contenuto dell’area puntata da “kernel” con la nuova configurazione. Lo stream di configurazione è costituito da un array di 51 byte: i primi 49 rappresentano i pesi di un kernel 7x7, gli ultimi due byte rappresentano rispettivamente i valori di somma e offset.

I valori relativi ai pesi sono memorizzati come variabili di tipo “double”, mentre il filtro attende pesi interi con valori a 8 bit da -128 a 127. Per trasformare i pesi da fixed point a interi a 8 bit, vengono moltiplicati per una potenza di due (2 ^ bit_shift), e troncati tramite un cast a “signed char”. La somma del kernel sarà moltiplicata per lo stesso numero, consentendo di mantenere il rapporto tra i pesi e la somma. Aumentando “bit_shift”, aumenterà quindi la precisione, a scapito però del range di numeri che sarà possibile inserire.

Della funzione “convolution_filter_configure” sono state realizzate due implementazioni differenti, una per il filtro tradizionale, l’altra per quello separabile, realizzato da Marco

Rossini. Nel file di intestazione è possibile scegliere quale dei due filtri utilizzare, specificandolo tramite una direttiva “#define”. Le due implementazioni differiscono principalmente per il modo in cui vanno a impostare i pesi del kernel. Nel caso del tradizionale, viene creato un kernel 7x7 a partire da uno di dimensioni variabili allineando quest’ultimo al centro del kernel 7x7; nel caso di quello separabile, il kernel di dimensioni variabili viene allineato con la prima riga e prima colonna.

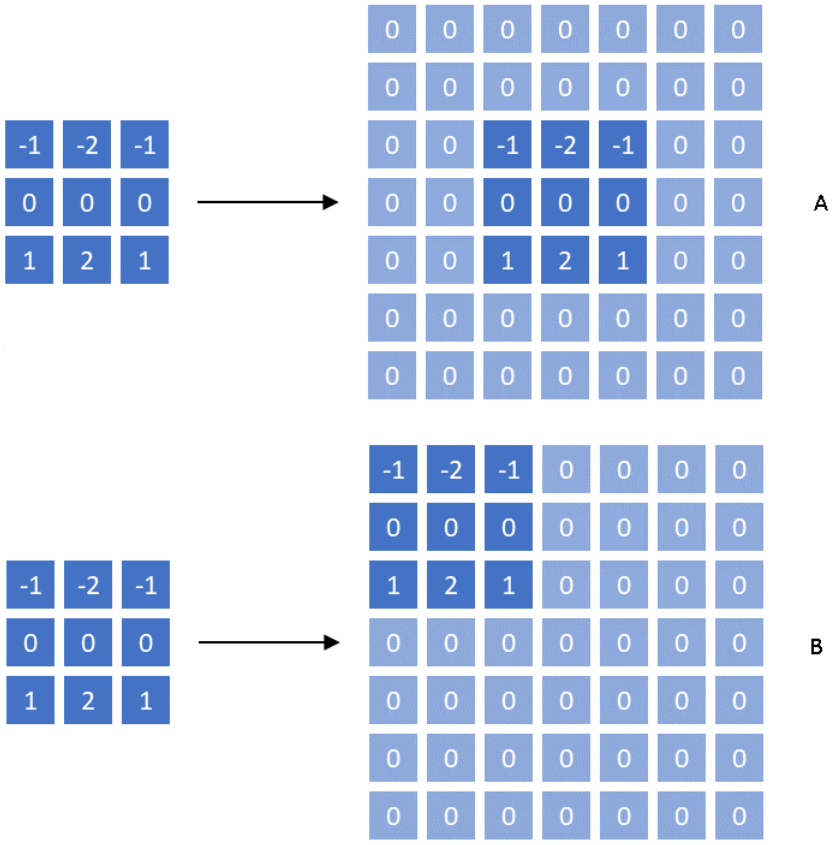


Figura 18: Configurazione dei pesi del kernel: tradizionale (A) e separabile (B)

```
#if defined(TRADITIONAL)
int convolution_filter_configure(kernel_config *config)
{
	int length, i, j;
	char filter_stream[KERNEL_HEIGHT * KERNEL_WIDTH + END_PARAMS_TRAD];
	double temp;

	if (config->height > KERNEL_HEIGHT
			|| config->height < 0
			|| config->width > KERNEL_WIDTH
			|| config->width < 0)
		return XST_FAILURE;

	//filter_stream initialization
	for (i = 0; i < KERNEL_HEIGHT * KERNEL_WIDTH; i++)
		filter_stream[i] = 0;

	//filter_stream weights setup
	for (i = 0; i < config->height; i++)
		for (j = 0; j < config->width; j++)
		{
			temp = config->weights[i * config->width + j] * (1 << config->bit_shift);

			if (temp > 127 || temp < -128)
			{
				xil_printf("[CONV FILTER] Weight [%d,%d] is out of range\n",i,j);
				return CONV_FILTER_FAILURE;
			}
			filter_stream[(i+(KERNEL_HEIGHT - config->height) / 2) * KERNEL_WIDTH + j
				+ (KERNEL_WIDTH - config->width) / 2] = (signed char)temp;
		}

	//additional parameters setup
	temp = config->sum * (1 << config->bit_shift);
	if (temp > 127 || temp < -128)
	{
		xil_printf("[CONV FILTER] Sum is out of range\n");
		return CONV_FILTER_FAILURE;
	}

	filter_stream[KERNEL_WIDTH * KERNEL_HEIGHT] = (signed char)temp;
	filter_stream[KERNEL_WIDTH * KERNEL_HEIGHT + 1] = config->offset;

	//axilite write
	length = XConvolution_filter_Write_kernel_config_V_Bytes
		(&filter_trad, 0, filter_stream, KERNEL_HEIGHT * KERNEL_WIDTH + END_PARAMS_TRAD);

	//check length
	if (length != KERNEL_HEIGHT * KERNEL_WIDTH + END_PARAMS_TRAD)
	{
		xil_printf("[CONV FILTER] An error occurred\n");
		return CONV_FILTER_FAILURE;
	}

	length = XSep_convolution_filter_Write_kernel_config_V_Bytes
	    (&filter_sep, 0, neutral_kernel_sep, KERNEL_HEIGHT * KERNEL_WIDTH + END_PARAMS_SEP);

	//check length
	if (length != KERNEL_HEIGHT * KERNEL_WIDTH + END_PARAMS_SEP)
	{
		xil_printf("[CONV FILTER] An error occurred\n");
		return CONV_FILTER_FAILURE;
	}

	copy_kernel_config(config, &kernel);

	return CONV_FILTER_SUCCESS;
}
#endif
```

```
#if defined(SEPARABLE)
int convolution_filter_configure(kernel_config *config)
{
	int length, i, j;
	char filter_stream[KERNEL_HEIGHT * KERNEL_WIDTH + END_PARAMS_SEP];
	double temp;

	if (config->height > KERNEL_HEIGHT
			|| config->height < 0
			|| config->width > KERNEL_WIDTH
			|| config->width < 0)
		return XST_FAILURE;

	//filter_stream initialization
	for (i = 0; i < KERNEL_HEIGHT * KERNEL_WIDTH; i++)
		filter_stream[i] = 0;

	//filter_stream weights setup
	for (i = 0; i < config->height; i++)
		for (j = 0; j < config->width; j++)
		{
			temp = config->weights[i * config->width + j] * (1 << config->bit_shift);

			if (temp > 127 || temp < -128)
			{
				xil_printf("[CONV FILTER] Weight [%d,%d] is out of range\n",i,j);
				return CONV_FILTER_FAILURE;
			}

			filter_stream[i * KERNEL_WIDTH + j] = (signed char)temp;
		}

	//additional parameters setup
	temp = config->sum * (1 << config->bit_shift);
	if (temp > 127 || temp < -128)
	{
		xil_printf("[CONV FILTER] Sum is out of range\n");
		return CONV_FILTER_FAILURE;
	}

	filter_stream[KERNEL_WIDTH * KERNEL_HEIGHT] = (signed char)temp;
	filter_stream[KERNEL_WIDTH * KERNEL_HEIGHT + 1] = config->offset;

	length = XSep_convolution_filter_Write_kernel_config_V_Bytes
			(&filter_sep, 0, filter_stream, KERNEL_HEIGHT * KERNEL_WIDTH +
END_PARAMS_SEP);

	//check length
	if (length != KERNEL_HEIGHT * KERNEL_WIDTH + END_PARAMS_SEP)
	{
		xil_printf("[CONV FILTER] An error occurred\n");
		return CONV_FILTER_FAILURE;
	}

	length = XConvolution_filter_Write_kernel_config_V_Bytes
	  (&filter_trad, 0, neutral_kernel_trad, KERNEL_HEIGHT * KERNEL_WIDTH + END_PARAMS_TRAD);

	//check length
	if (length != KERNEL_HEIGHT * KERNEL_WIDTH + END_PARAMS_TRAD)
	{
		xil_printf("[CONV FILTER] An error occurred\n");
		return CONV_FILTER_FAILURE;
	}

	copy_kernel_config(config, &kernel);

	return CONV_FILTER_SUCCESS;
}
#endif
```

La funzione “init_convolution_filter” inizializza i filtri, rappresentati a livello applicativo dalle variabili globali “filter_trad” e “filter_sep”, il cui tipo è definito nei driver generati automaticamente, nel BSP. Viene chiamata durante la fase di inizializzazione dell’applicazione, insieme alle altre funzioni di inizializzazione degli altri ip core su cui sono necessarie letture e/o scritture dal software applicativo. Dopo l’inizializzazione, entrambi i filtri vengono configurati automaticamente con un kernel neutrale di dimensioni 1x1, somma 1, offset 0 e 0 bit decimali.

```
int convolution_filter_init(void)
{
	int status;

	status = XSep_convolution_filter_Initialize(&filter_sep,
		XPAR_XSEP_CONVOLUTION_FILTER_0_DEVICE_ID);
	status = XConvolution_filter_Initialize(&filter_trad,
		XPAR_XCONVOLUTION_FILTER_0_DEVICE_ID);

	if (status != XST_SUCCESS)
	{
		xil_printf("An error occurred while initializing the convolution filter\n");
		return status;
	}

	temp_config = (kernel_config*)malloc(sizeof(kernel_config));

	temp_config->height = 1;
	temp_config->width = 1;
	temp_config->sum = 1;
	temp_config->offset = 0;
	temp_config->bit_shift = 0;
	temp_config->weights = (double*)malloc(sizeof(double));
	temp_config->weights[0] = 1;

	status = convolution_filter_configure(temp_config);

	if (status != CONV_FILTER_SUCCESS)
	{
		xil_printf("An error occurred while initializing the convolution filter\n");
		return XST_FAILURE;
	}

	return XST_SUCCESS;
}
```

4.2 - Cenni sull'implementazione LWIP del protocollo TCP

LWIP [12] (LightWeight IP) è una implementazione (in linguaggio C) cross-platform e open source dello stack TCP/IP, originariamente sviluppata da Adam Dunkels, largamente utilizzata nell'ambito dei sistemi embedded, per via del ridotto utilizzo di risorse.

Per quanto concerne il protocollo TCP, tutte le ricezioni di dati, sia in fase di handshake iniziale, sia in fase di comunicazione, avvengono tramite un meccanismo di callback, che consente di ricevere dati senza effettuare chiamate bloccanti. Prima di ricevere un dato infatti, è necessario specificare una funzione di callback, questa funzione verrà invocata automaticamente al momento della ricezione dei dati.

Tra le numerose funzioni offerte, vengono riportate in particolare le più importanti:

- `struct tcp_pcb * tcp_new(void)`

Viene creato e restituito il PCB (Protocol Control Block), e una connessione nello stato "Closed". Il PCB contiene tutte le informazioni relative alla connessione.

- `err_t tcp_bind(struct tcp_pcb *pcb, ip_addr_t *ipaddr, u16_t port)`

Viene effettuato il binding (esplicito) del PCB specificato a un indirizzo e porta locali.

- `struct tcp_pcb * tcp_listen(struct tcp_pcb *pcb)`

Viene deallocato il PCB passato come argomento, che rappresentava una connessione chiusa, e viene inizializzato un nuovo PCB in stato di "listen".

- `void tcp_accept(struct tcp_pcb *pcb, tcp_accept_fn accept)`

Associa al PCB specificato una funzione da chiamarsi ogni volta che un host effettua una richiesta di connessione.

- `void tcp_recv(struct tcp_pcb *pcb, tcp_recv_fn recv)`

Associa al PCB specificato, similmente alla funzione "tcp_accept", una funzione da chiamarsi ogni volta che vengono ricevuti dati.

- `err_t tcp_write(struct tcp_pcb *pcb, const void *arg, u16_t len, u8_t apiflags)`

Invia dati sul canale in uscita della connessione associata al PCB specificato. I dati da inviare sono i primi "len" byte nell'area di memoria puntata da "arg".

- `err_t tcp_close(struct tcp_pcb *pcb)`

Viene chiusa la connessione associata al PCB specificato.

4.3 - Modulo “tcp_connection”

4.3.1 - Precedente Implementazione

Il modulo “tcp_connection” si occupa della comunicazione via TCP con il client. Il modulo era già parzialmente implementato, fornendo un servizio dummy di esempio, utilizzando l’implementazione dello stack TCP/IP fornita da LWIP. Della precedente implementazione sono state riutilizzate le funzioni delle fasi di apertura e chiusura della connessione, vale a dire “start_tcp”, “connection_accept” e “close_conn”, discusse in dettaglio nella tesi di Mattia Bernasconi [1], il cui funzionamento è di seguito brevemente riassunto.

“start_tcp” viene chiamata in fase di inizializzazione dell’applicazione, crea il PCB, effettua il binding all’indirizzo locale 192.168.1.50, e alla porta 2101. Mette il PCB in stato di “listen” e specifica la funzione “connection_accept” come callback da chiamarsi al momento della ricezione di richieste di connessione.

```
int start_tcp(void)
{
	xil_printf("[TCP] Starting TCP server on port %d\n", TCP_PORT);

	// Create a new TCP Protocol Cotrol Block abbreviated as PCB.
	pcb_tcp = tcp_new();

	// Bind the PCB to a Port and any IP address.
	tcp_bind(pcb_tcp, IP_ADDR_ANY, TCP_PORT);

	// Put the PCB and TCP connection on Board in listening state
	pcb_tcp = tcp_listen(pcb_tcp);

	xil_printf("[TCP] Listening on port %d\n\n", TCP_PORT);

	// This function specifies the callback function that will be called when a client asks
for connection with board
	tcp_accept(pcb_tcp, connection_accept);

	return XST_SUCCESS;
}
```

“connection_accept” accetta la richiesta di connessione, e specifica “callback_tcp_request” come funzione di callback da chiamare al momento della ricezione di nuovi dati. La funzione “reset_req_buffer” inizializza il buffer delle richieste e lo pone nello stato di attesa del prossimo header. Il funzionamento del buffer delle richieste verrà mostrato in dettaglio al paragrafo successivo.

```
static err_t connection_accept(void *arg, struct tcp_pcb *pcb_tcp, err_t err)
{
	print_ip("\n[TCP] Connection accepted from ", &pcb_tcp->remote_ip);

	LWIP_UNUSED_ARG(arg);
	LWIP_UNUSED_ARG(err);

	tcp_setprio(pcb_tcp, TCP_PRIO_MIN);

	// request system setup
	reset_req_buffer();
	xil_printf("[TCP] Now waiting for a request header\n", req_len, req_type);

	// Specifies the function to call, on reception of data on TCP
	tcp_recv(pcb_tcp, callback_tcp_request);

	tcp_err(pcb_tcp, NULL);
	tcp_poll(pcb_tcp, NULL, 4);

	return ERR_OK;
}
```

La funzione “close_conn” disabilita tutti i callback e chiude la connessione:

```
static void close_conn(struct tcp_pcb *pcb_tcp)
{
	tcp_arg(pcb_tcp, NULL);
	tcp_sent(pcb_tcp, NULL);
	tcp_recv(pcb_tcp, NULL);
	tcp_close(pcb_tcp);
	print_ip("[TCP] Connection closed with client ", &pcb_tcp->remote_ip);
}
```

4.3.2 - Protocollo di comunicazione tra client e server

La comunicazione tra cliente e servitore avviene seguendo un semplice protocollo definito ad hoc. Ogni richiesta si compone di un header e un body, l'header specifica il tipo di richiesta (1 byte) e la lunghezza del body (2 byte). Tale lunghezza può anche essere nulla, indicando una richiesta senza body. Il body contiene tutti i valori necessari al completamento della richiesta. Il funzionamento del servitore può essere schematizzato dal seguente diagramma di stato.

All'apertura di una connessione il server si trova nello stato "Attesa Header", in cui permane fino al ricevimento dell'header, passando nello stato "Header Ricevuto". In questo stato è noto il tipo di richiesta e la lunghezza del body da ricevere. Il server rimane in questo stato finché non ha ricevuto tutti i byte del body, in qualsiasi momento è possibile passare allo stato "Disconnesso". Terminata la ricezione del body, viene eseguita la richiesta, e si attende la ricezione del prossimo header.

Vista l'assenza di controllo sul momento in cui avviene la lettura dei dati, questi potrebbero arrivare in qualsiasi momento e non necessariamente tutti in un'unica esecuzione della funzione "callback_tcp_request". È necessario pertanto memorizzare temporaneamente i byte ricevuti in un buffer "req_buff", allocato dinamicamente di lunghezza "req_len", pari al numero di byte da cui la richiesta è composta. La variabile "req_index" indica la posizione a cui scrivere il prossimo byte all'interno del buffer, e "req_type" indica il tipo di richiesta che il servitore sta attendendo in questo momento.

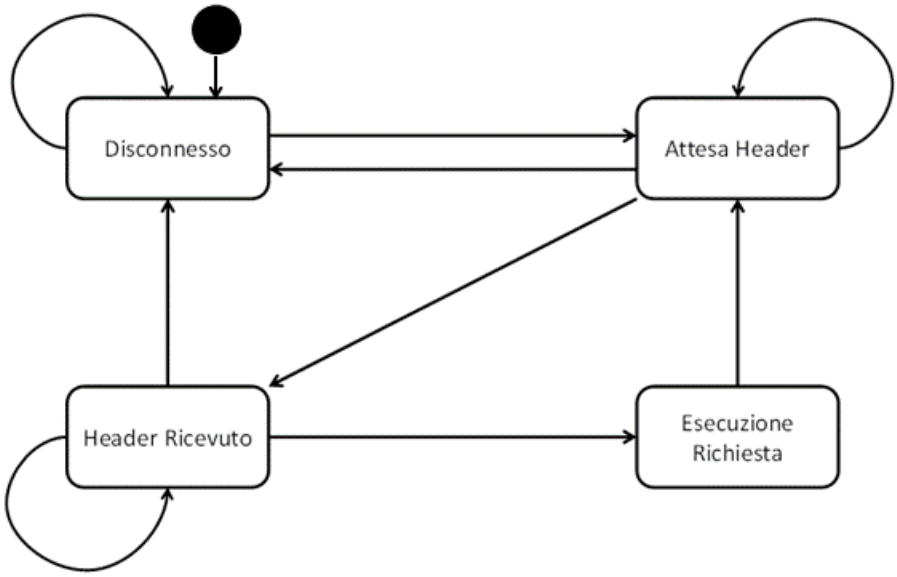


Figura 19: Diagramma a stati del servitore

Il buffer delle richieste viene resettato (e inizializzato) utilizzando la funzione "reset_req_buffer". Questa funzione de-alloca se necessario la memoria occupata dal precedente buffer, imposta la lunghezza attesa a HEADER_LEN (pari a 3), alloca la memoria necessaria, imposta l'indice su cui scrivere il prossimo byte a 0, e imposta il tipo di richiesta a REQUEST_NONE. Quando "req_type" assume tale valore, significa che il server è in attesa di un header e pertanto non conosce il tipo di richiesta, in tutti gli altri casi indica che il server ha ricevuto un header, conosce quindi il tipo di richiesta e la sua lunghezza. La ricezione di un header è considerata come una richiesta di tipo REQUEST_NONE, e viene gestita come tutte le altre richieste al momento in cui il buffer è pieno.

```
void reset_req_buffer()
{
	if (req_buffer != NULL)
		free(req_buffer);

	req_len = HEADER_LEN;
	req_buffer = (unsigned char*)malloc(req_len);
	req_index = 0;
	req_type = REQUEST_NONE;
}
```

La funzione "callback_tcp_request", che viene invocata ad ogni ricezione di dati, controlla che non si siano verificati errori di trasmissione, e che il buffer non sia vuoto, in tal caso, notifica il client della ricezione dei dati, utilizzando "tcp_recved". Ottiene i byte ricevuti, e li memorizza uno alla volta nel buffer delle richieste, "req_buff", controllando ad ogni inserimento la possibilità di eseguire richieste.

```
// This function gets called on reception of data on TCP.
static err_t callback_tcp_request(void *arg, struct tcp_pcb *pcb_tcp, struct pbuf *pbuffer,
err_t err)
{
	unsigned int len;
	unsigned char *pc;

	// Check if no error has occurred and packet buffer is not empty
	if (err == ERR_OK && pbuffer != NULL)
	{
		tcp_recved(pcb_tcp, pbuffer->tot_len);

		// Get the pointer to the Payload area of received TCP packet
		pc = (unsigned char *) pbuffer->payload;

		// Get the length of the data payload
		len = pbuffer->tot_len;

		// Debug print
		xil_printf("[TCP] Received %d bytes\n", len);

		// For each byte received:
		int i;
		for (i = 0; i < len; i++)
		{
			// Update the request buffer
			req_buffer[req_index++] = pc[i];

			// If the request is ready to be performed, perform it
			while (req_index >= req_len)
			{
				perform_request(pcb_tcp);
			}
		}

	}
	// Free the packet buffer
	pbuf_free(pbuffer);


	// Close connection if empty packet and no error
	if (err == ERR_OK && pbuffer == NULL)
	{
		close_conn(pcb_tcp);
	}

	return ERR_OK;
}
```

Nel momento in cui una richiesta è pronta per essere eseguita, viene invocata la funzione "perform_request", che gestisce le richieste dei seguenti tipi:

- REQUEST_NONE: il server ha ricevuto un header, e deve mettersi in attesa del body, aggiornando le strutture dati e lo stato.
- REQUEST_SET_KERNEL_SUM: il client richiede di modificare la somma del kernel attualmente impostato.
- REQUEST_SET_KERNEL_OFF: il client richiede di modificare l'offset del kernel attualmente impostato.
- REQUEST_SET_KERNEL_WEIGHTS: il client richiede di modificare i pesi del kernel attualmente impostato.
- REQUEST_GET_KERNEL_CONFIG: il client richiede di conoscere la configurazione del filtro.
- REQUEST_SET_KERNEL_BIT_SHIFT: il client richiede di modificare i bit di precisione dei pesi.

Al termine dell'esecuzione di una richiesta (dopo aver ricevuto il body), il sistema viene messo in attesa del prossimo header.

```
void perform_request(struct tcp_pcb * pcb_tcp)
{
int retval = CONV_FILTER_FAILURE; //If the system has just finished receiving a partially
received request
	if (req_type == REQUEST_NONE)
	{
		req_type = req_buffer[0];
		if (req_type != REQUEST_NONE)
		{
			req_len = req_buffer[1] * 256 + req_buffer[2];
			free(req_buffer);
			req_buffer = (unsigned char*)malloc(req_len);
		}
		req_index = 0;
		xil_printf("[TCP] Header received, now waiting for a %d bytes request body\n",
req_len);
		return;
	}

	//Client requests to modify the kernel sum value
	else if (req_type == REQUEST_SET_KERNEL_SUM)
	{
		if (req_index >= 1){
			kernel_config *kernel = convolution_filter_get_kernel_config();
			kernel->sum = req_buffer[0];
			retval = convolution_filter_configure(kernel);
		}
		reset_req_buffer();

		send_ack(pcb_tcp, retval);
	}

	//Client requests to modify the kernel offset value
```

```
	else if (req_type == REQUEST_SET_KERNEL_OFF)
	{
		if (req_index >= 1){
			kernel_config *kernel = convolution_filter_get_kernel_config();
			kernel->offset = req_buffer[0];
			retval = convolution_filter_configure(kernel);
		}
		reset_req_buffer();

		send_ack(pcb_tcp, retval);
	}

	//Client requests to modify the kernel weights
	else if (req_type == REQUEST_SET_KERNEL_WEIGHTS)
	{
		kernel_config *kernel = convolution_filter_get_kernel_config();

		if (req_index >= 3){
			kernel->height = req_buffer[0];
			kernel->width = req_buffer[1];
		}

		if (req_index - 2 >= kernel->height * kernel->width)
		{
			int i;
			free(kernel->weights);
			kernel->weights = (double*)malloc(sizeof(double) * kernel->height *
kernel->width);
			for (i = 0; i < kernel->height * kernel->width; i++)
			{
				kernel->weights[i] = char_to_double(req_buffer[2 * i + 2],
req_buffer[2 * i + 3]);
			}
		}

		retval = convolution_filter_configure(kernel);
		reset_req_buffer();
		send_ack(pcb_tcp, retval);
	}

	//Client requests to get the current kernel configuration
	else if (req_type == REQUEST_GET_KERNEL_CONFIG)
	{
		tcp_send_kernel_config(pcb_tcp);
		reset_req_buffer();
	}

	//Client requests to get the current kernel bit shift approximation
	else if (req_type == REQUEST_SET_KERNEL_BIT_SHIFT)
	{
		if (req_index >= 1){
			kernel_config *kernel = convolution_filter_get_kernel_config();
			kernel->bit_shift = req_buffer[0];
			retval = convolution_filter_configure(kernel);
		}
		reset_req_buffer();

		send_ack(pcb_tcp, retval);
	}

	//In case of error: resets the buffer and waits for a new request
	else
	{
		reset_req_buffer();
	}

	xil_printf("[TCP] Now waiting for reception of a request header\n");

}
```

All'interno della funzione "perform_request" vengono chiamate altre funzioni di utilità, per rendere il codice più pulito e riutilizzabile. Nello specifico, "tcp_send_kernel_config" invia al client la configurazione del kernel, "send_ack" invia al client un codice di errore, per notificarlo dell'avvenuto completamento dell'operazione, "char_to_double" e "double_to_char" convertono due char in un double e viceversa. Ciò avviene in quanto si è scelto di trasmettere i valori fixed-point mediante due byte: il primo con segno, che rappresenta la parte intera arrotondata per difetto, il secondo senza segno, che rappresenta la differenza tra il valore da rappresentare e il primo byte, moltiplicata per 256 e arrotondata per difetto.

```
void tcp_send_kernel_config(struct tcp_pcb *pcb_tcp)
{
	kernel_config * config;
	int i, len;
	char * data;

	config = convolution_filter_get_kernel_config();
	len = 2 + 2 * config->height * config->width + 2;
	data = (char*)malloc(len);

	data[0] = config->height;
	data[1] = config->width;

	xil_printf("%d %d\n", data[0], data[1]);

	for (i = 0; i < config->height * config->width; i++)
	{
		double_to_char(config->weights[i], &data[2 * i + 2], &data[2 * i + 3]);
	}

	xil_printf("\n");
	data[2 * i + 2] = config->sum;
	data[2 * i + 3] = config->offset;
	data[2 * i + 4] = config->bit_shift;

	tcp_write(pcb_tcp, data, len, TCP_WRITE_FLAG_COPY);

	free(data);
}
```

```
void double_to_char(double value, char *a, char *b)
{
	*a = (signed char)value;
	*b = (unsigned char)((int)(value * 256) % 256);

	if (value < 0 && *b != 0)
	{
		(*a)--;
	}
}
```

```
double char_to_double(char a, char b)
{
	double value;

	value = (signed char)a;
	value += ((double)((unsigned char)b)) / (double)256;

	return value;
}
```

```
void send_ack(struct tcp_pcb *pcb_tcp, int err_code)
{
	char *data;
	int size = 1;
	// Open for future changes, err_code size may be variable

	data = (char*)malloc(size);
	data[0] = (char)err_code;
	xil_printf("[TCP] Sending ack code: %d\n", data[0]);
	tcp_write(pcb_tcp, data, size, TCP_WRITE_FLAG_COPY);
	free(data);
}
```

Capitolo 5 - Software lato Cliente

Il client di configurazione è stato realizzato in linguaggio C, in due versioni, a causa della differenza delle API di rete, per Linux/Mac e per Windows. Il client offre la possibilità di impostare i parametri di configurazione del filtro (dimensione, somma, offset, pesi e precisione), nonché la possibilità di ricevere la configurazione corrente, e ripristinare il filtro alla configurazione di default.

Oltre al modulo "main", il software si compone di due moduli "connection" e "control": il primo funge da wrapper delle API di rete del sistema operativo: socket unix per Linux/Mac e winsock2 [15] per Windows; il secondo utilizza queste funzionalità per implementare il protocollo di comunicazione tra client e server (board).

5.1 - Modulo "connection"

Questo modulo consente di mantenere il modulo "control" indipendente dal sistema operativo utilizzato, in quanto per inviare e ricevere dati, esso utilizza esclusivamente le astrazioni fornite dal modulo "connection".

Nello specifico, nell'header sono importate le librerie appropriate, e dichiarati quattro prototipi di funzione, per apertura e chiusura della connessione, invio e ricezione di dati.

```
int start_connection();

int close_connection();

int send_data(char* data, int len);

int recv_data(char* data, int len);
```

Nello specifico, "start_connection" inizializza le strutture dati necessarie per mantenere le informazioni di indirizzo e porta, crea una nuova socket, ed effettua un tentativo di connessione alla board (all'indirizzo 192.168.1.50 e porta 2101). Restituisce un codice di errore per notificare o meno il successo dell'operazione.

Nelle pagine successive sono riportate le due implementazioni (Linux/Mac) e (Windows) della funzione "start_connection".

```
int sd;

int start_connection() // LINUX - MAC
{
	struct sockaddr_in servaddr;
	struct in_addr addr;
	struct hostent * host;
	int status;

#ifdef DEBUG
	printf("Connecting with the board...\n");
#endif

	memset((char *)&servaddr, 0, sizeof(struct sockaddr_in));
	servaddr.sin_family = AF_INET;

#ifdef DEBUG
	printf("sin_family initialized\n");
#endif

	host = gethostbyname(BOARD_INET_ADDR);
	if (host == NULL)
	{
		printf("No host %s found\n", BOARD_INET_ADDR);
		return FAILURE;
	}

#ifdef DEBUG
	printf("struct hostent initialized\n");
#endif

	servaddr.sin_addr.s_addr = ((struct in_addr *) (host->h_addr))->s_addr;
	servaddr.sin_port = htons(BOARD_PORT);

#ifdef DEBUG
	printf("struct sockaddr_in initialized\n");
#endif

	sd = socket(AF_INET, SOCK_STREAM, 0);
	if (sd < 0)
	{
		printf("An error occurred during socket creation\n");
		return FAILURE;
	}
#ifdef DEBUG
	printf("Socket created: %d\n", sd);
#endif

	if ((status = connect(sd, (struct sockaddr *)&servaddr, sizeof(struct sockaddr))) < 0)
	{
		printf("Could not connect with the board %d\n", errno);
		close(sd);
		return FAILURE;
	}
#ifdef DEBUG
	printf("Connection established\n");
#endif

	return SUCCESS;
}
```

```
SOCKET ConnectSocket;

int start_connection() // WINDOWS
{
	WSADATA wsaData;
	struct addrinfo *result = NULL,
			hints;
	int iResult;

	ConnectSocket = INVALID_SOCKET;

	// Initialize Winsock
	iResult = WSAStartup(MAKEWORD(2,2), &wsaData);
	if (iResult != 0) {
		printf("WSAStartup failed with error: %d\n", iResult);
		return FAILURE;
	}

#ifdef DEBUG
	printf("Windows socket initialized successfully\n");
#endif

	ZeroMemory( &hints, sizeof(hints) );
	hints.ai_family = AF_INET;
	hints.ai_socktype = SOCK_STREAM;
	hints.ai_protocol = IPPROTO_TCP;

#ifdef DEBUG
	printf("Socket hints zeroed successfully\n");
#endif

	// Resolve the server address and port
	iResult = getaddrinfo(BOARD_INET_ADDR, BOARD_PORT, &hints, &result);
	if ( iResult != 0 ) {
		printf("getaddrinfo failed with error: %d\n", iResult);
		WSACleanup();
		return FAILURE;
	}

#ifdef DEBUG
	printf("getaddrinfo executed successfully\n");
#endif

	// Create a SOCKET for connecting to server
	ConnectSocket = socket(result->ai_family, result->ai_socktype,
			result->ai_protocol);
	if (ConnectSocket == INVALID_SOCKET) {
		printf("socket failed with error: %d\n", WSAGetLastError());
		WSACleanup();
		return FAILURE;
	}

#ifdef DEBUG
	printf("Windows Socket created\n");
#endif

	// Connect to server.
	iResult = connect(ConnectSocket, result->ai_addr, (int)result->ai_addrlen);
	if (iResult == SOCKET_ERROR) {
		closesocket(ConnectSocket);
		ConnectSocket = INVALID_SOCKET;
		return FAILURE;
	}

#ifdef DEBUG
	printf("Windows Socket connected\n");
#endif

	freeaddrinfo(result);
```

```
        if (ConnectSocket == INVALID_SOCKET) {
                printf("Unable to connect to server\n");
                WSACleanup();
                return FAILURE;
        }

        return SUCCESS;
}
```

Dopo aver stabilito con successo una connessione con la board, è possibile inviare e ricevere dati utilizzando le funzioni generiche “send_data” e “recv_data”, che rispettivamente inviano e ricevono un certo numero di byte. Nel caso non siano disponibili i dati richiesti, la chiamata a “recv_data” è bloccante. Entrambe le funzioni restituiscono un codice di errore per notificare dell’avvenuto invio o ricezione dei dati.

È necessario, come per la funzione “connection_start”, fornire due implementazioni diverse per ciascuna di queste funzioni, una basata su socket unix e l’altra basata su winsock2. Nelle pagine seguenti, viene riportato il codice di entrambe le versioni delle funzioni “send_data” e “recv_data”.

```
int send_data(char * data, int len) // LINUX - MAC
{
    if (write(sd, data, len)<0)
    {
        printf("An error occurred while sending data\n");
        return FAILURE;
    }

    #ifdef DEBUG
    printf("Sent %d bytes of data\n", len);
    #endif

    return SUCCESS;
}

int recv_data(char * data, int len) // LINUX - MAC
{
    if (read(sd, data, len)<0)
    {
        printf("An error occurred while receiving data\n");
        return FAILURE;
    }

    #ifdef DEBUG
    printf("Received %d bytes of data\n", len);
    #endif

    return SUCCESS;
}
```

```
int send_data(char * data, int len) // WINDOWS
{
    int res;

    // Send an initial buffer
    res = send( ConnectSocket, data, len, 0 );
    if (res == SOCKET_ERROR)
    {
        printf("send failed with error: %d\n", WSAGetLastError());
        return FAILURE;
    }
#ifdef DEBUG
    printf("Sent %d bytes of data\n", res);
#endif
    return SUCCESS;
}

int recv_data(char * data, int len) // WINDOWS
{
    int res;
    res = recv(ConnectSocket, data, len, 0);
    if (res < 0)
    {
        printf("recv failed with error: %d\n", WSAGetLastError());
        return FAILURE;
    }
    if (res == 0)
    {
        printf("connection closed\n");
        return FAILURE;
    }
#ifdef DEBUG
    printf("Received %d bytes of data\n", res);
#endif
    return SUCCESS;
}
```

La funzione “close_connection” chiude la connessione e libera le risorse di sistema ad essa associate. Anche per questa funzione sono necessarie due versioni, in base al sistema operativo utilizzato.

```
int close_connection() // LINUX - MAC
{
    int retval;

    #ifdef DEBUG
    printf("Disconnecting from the board...\n");
    #endif

    retval = close(sd);

    return retval;
}
```

```
int close_connection() // WINDOWS
{
    int retval;

    #ifdef DEBUG
    printf("Disconnecting from the board...\n");
    #endif

    // shutdown the connection since no more data will be sent
    retval = shutdown(ConnectSocket, SD_SEND);
    if (retval == SOCKET_ERROR)
    {
        printf("shutdown failed with error: %d\n", WSAGetLastError());
        closesocket(ConnectSocket);
        WSACleanup();
        return FAILURE;
    }

    // cleanup
    closesocket(ConnectSocket);
    WSACleanup();

    return SUCCESS;
}
```

5.2 - Modulo “control”

Il modulo di controllo si occupa di implementare il semplice protocollo di comunicazione tra client e board, precedentemente definito. Grazie alle funzionalità offerte dal modulo “connection” è completamente indipendente dal sistema operativo utilizzato, favorendo la portabilità del codice.

Sono definite inoltre due funzioni per l’invio e ricezione di valori fixed-point, che trasformano un double nella rappresentazione a due byte (descritta precedentemente) e viceversa, analoghe a quelle presenti lato servitore.

La funzione “read_loop” avvia un loop di letture da standard input, interpreta il comando inserito, e invia conseguentemente una richiesta al server. Attende infine una risposta, e nel caso di errore notificherà l’utente.

I comandi disponibili sono i seguenti (possibilità di estensione):

- `set sum|off|weights|bit_shift <value>`

Imposta i valori di somma, offset, pesi e precisione. Nel caso dei pesi, attende l’inserimento di 2 valori rappresentanti le due dimensioni del kernel H e W, e H*W valori rappresentanti i pesi.

- `Reset`

Ripristina la configurazione di default: kernel 1x1 identità, somma 1, offset 0 e precisione unitaria.

- `Print`

Riceve e stampa la configurazione corrente del filtro.

- `Exit`

Disconnette il client e termina l’applicazione di configurazione.

Capitolo 6 - Risultati Sperimentali

La possibilità di configurare il filtro dall'esterno introduce dei cambiamenti nella logica del filtro, ciò influisce sulle performance e sulle risorse utilizzate. Sono quindi state effettuate sintesi di prova variando le dimensioni del kernel, da un minimo di 3x3 ad un massimo di 17x17, aumentando ogni volta entrambe le dimensioni di 2 unità.

Di queste sintesi sono stati raccolti alcuni dati significativi, ovvero:

- Timing: tempo medio necessario al completamento di una iterazione. Per garantire il corretto funzionamento dell'hardware sintetizzato, questo valore deve mantenersi al di sotto di un valore di riferimento (clock target), fissato a 41,67 ns. Questo valore è pari al periodo del clock a cui verrà collegato il filtro.
- Total Latency: numero di cicli di clock necessari al completamento dell'operazione.
- Loop Latency: numero di cicli di clock necessari al completamento della pipeline.
- Loop Trip Count: numero di iterazioni nel loop, idealmente pari al prodotto delle dimensioni dell'immagine, incrementate della metà delle rispettive dimensioni del kernel. (Per un kernel 7x7 e un'immagine 640x480, dovrebbe risultare 643 * 483 = 310569)
- Iteration Latency: numero di step in cui viene suddivisa la singola iterazione.
- BRAM, DSP48, FF, LUT: risorse hardware impiegate. Viene riportato anche il relativo valore percentuale, relativo alle risorse disponibili.

Kernel size	3x3	5x5	7x7	9x9	11x11	13x13	15x15	17x17
Timing	20.69	22.81	25.06	27.31	27.31	29.55	29.55	31.8
Total Latency	308348	309472	310598	311726	312855	313987	315120	316256
Loop Latency	308346	309470	310596	311724	312853	313985	315118	316254
Loop Trip Count	308321	309444	310569	311696	312825	313956	315089	316224
Iteration Latency	27	28	29	30	30	31	31	32
BRAM	2	4	6	8	10	12	14	16
DSP48	9	25	49	81	121	169	255	289
FF	1138	1489	1968	2579	3255	4162	5510	7125
LUT	1321	1491	1726	2103	2570	2893	4287	5933
BRAM %	0.71	1.43	2.14	2.86	3.57	4.29	5.00	5.71
DSP48 %	4.09	11.36	22.27	36.82	55.00	76.82	115.91	131.36
FF %	1.07	1.40	1.85	2.42	3.06	3.91	5.18	6.70
LUT %	2.48	2.80	3.24	3.95	4.83	5.44	8.06	11.15

Come previsto, i tempi e le risorse impiegate aumentano all'incrementare delle dimensioni massime del kernel. Sulla sintesi del filtro di convoluzione sono imposti dei forti vincoli relativi al timing, che non deve superare i 41,67 ns, ovvero il periodo di un ciclo di clock a 24 MHz, tuttavia, sebbene ciò costituisca un problema nella sintesi di filtri con kernel di dimensione maggiore, il principale problema di scalabilità del filtro è dovuto ai limiti delle risorse presenti sul FPGA. In particolare si osserva come già un filtro 7x7 da solo utilizzi ben il 23% dei DSP presenti sul FPGA, e come un filtro di dimensioni maggiori o uguali a 15x15 non risulti nemmeno integrabile sulla logica programmabile, in quanto necessiterebbe di un numero di DSP maggiore rispetto a quanti ne siano presenti sul FPGA. Bisogna inoltre tenere conto dei BSP utilizzati da eventuali altri ip core integrati nel design, per cui se il filtro utilizzasse una percentuale eccessivamente prossima al 100% dei BSP presenti, altri ip core potrebbero non riuscire ad essere integrati correttamente.

L'utilizzo di DSP è dovuto alla presenza di numerose operazioni MAC (Multiply and Accumulate operation), tuttavia, tramite una direttiva, è possibile mappare queste operazioni MAC su delle tabelle LUT, senza utilizzare i DSP. Questo incide sull'utilizzo delle LUT, ma rende il design più scalabile, in quanto, in rapporto ai DSP, le LUT sono presenti in quantità superiori.

Per forzare la mappatura su LUT delle operazioni MAC si utilizza la direttiva [8]

```
#pragma HLS RESOURCE variable=temp core=Mul_LUT
```

Associate alla variabile "temp", su cui vengono ad ogni ciclo memorizzati i prodotti dei pixel per i pesi del kernel. Questo produce i risultati riportati nella seguente tabella:

Kernel size	3x3	5x5	7x7	9x9	11x11	13x13	15x15	17x17
Timing	18.27	20.39	22.64	24.89	24.89	27.13	27.13	29.38
Total Latency	308348	309472	310598	311726	312855	313987	315120	316256
Loop Latency	308346	309470	310596	311724	312853	313985	315118	316254
Loop Trip Count	308321	309444	310569	311696	312825	313956	315089	316224
Iteration Latency	27	28	29	30	30	31	31	32
BRAM	2	4	6	8	10	12	14	16
DSP48	0	0	0	0	0	0	0	0
FF	1138	1489	1968	2579	3255	4162	5510	7125
LUT	1983	3370	5421	8137	11409	15535	20894	27199
BRAM %	0.71	1.43	2.14	2.86	3.57	4.29	5.00	5.71
DSP48 %	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
FF %	1.07	1.40	1.85	2.42	3.06	3.91	5.18	6.70
LUT %	3.73	6.33	10.19	15.30	21.45	29.20	39.27	51.13

Come è osservabile, il tempo necessario al completamento dell'operazione è leggermente diminuito, i cicli di clock necessari, le BRAM e i FF impiegati, sono rimasti invariati. Differisce l'utilizzo di DSP, ora nullo, e di conseguenza l'utilizzo di LUT, che è ovviamente aumentato notevolmente. Ciononostante, se senza la direttiva il filtro con il kernel 17x17 arrivava ad impiegare oltre il 100% delle risorse disponibili, ora è perfettamente sintetizzabile, utilizzando poco più del 50% delle LUT.

La scalabilità è ancora un problema, ma è mitigato dal compromesso fornito dalla direttiva. Per migliorare ulteriormente l'impiego di risorse, potrebbe essere valutata la sintesi di un filtro separabile, discussa nella tesi di Marco Rossini. Ciò consentirebbe, nel caso in cui il kernel abbia rango pari a 1, di eseguire un numero di MAC pari al doppio del lato del kernel, non più al suo quadrato, diminuendo notevolmente il numero di operazioni da eseguire.

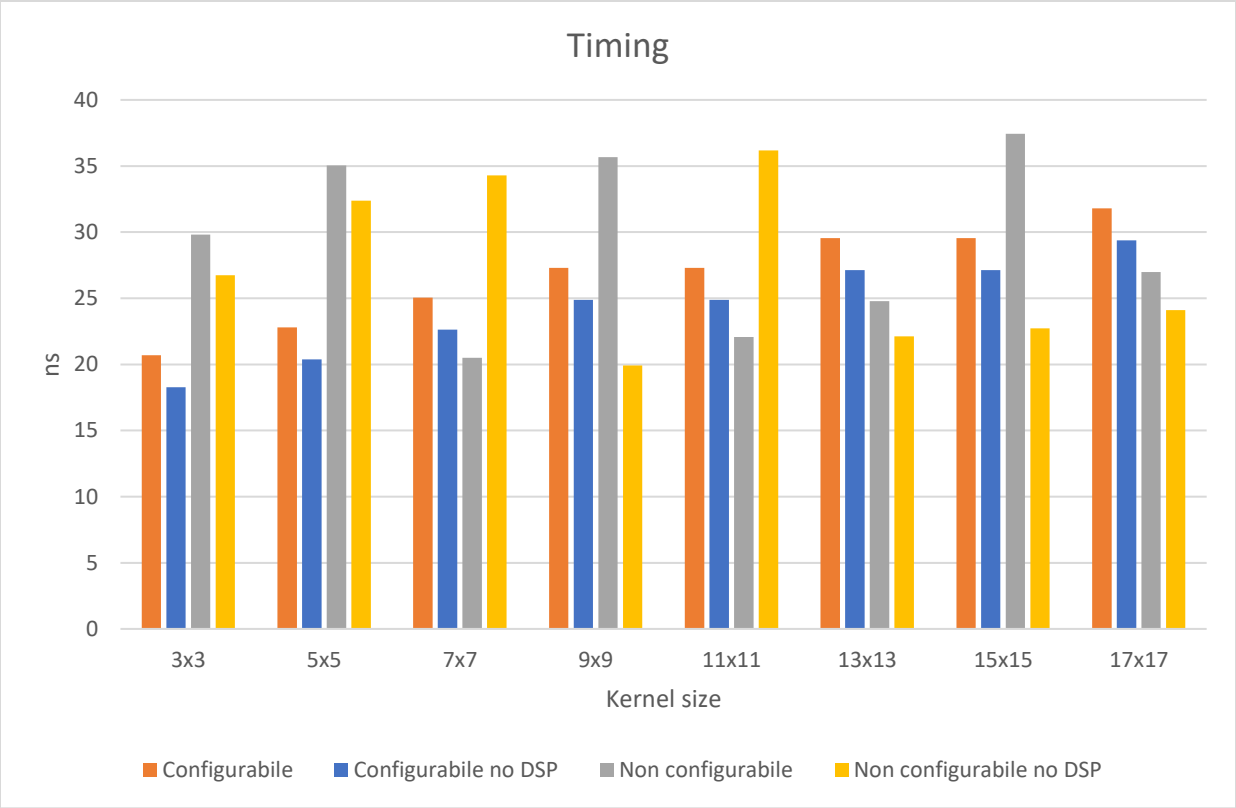
Per effettuare una stima dei costi della configurabilità del filtro, vengono presi in esame i dati relativi al filtro precedente, non configurabile. Per effettuare stime più accurate e ripetibili, il filtro è stato pre-configurato, in tutti i casi analizzati, con un kernel generato casualmente. Anche di questo filtro è stata realizzata una versione che limita l'utilizzo di DSP.

I risultati sono riportati nelle due seguenti tabelle.

Kernel size	3x3	5x5	7x7	9x9	11x11	13x13	15x15	17x17
Timing	29.82	35.04	20.51	35.67	22.07	24.79	37.44	26.99
Total Latency	308324	309447	310573	311699	312289	313690	315092	316228
Loop Latency	308322	309445	301571	311697	312287	313688	315090	316226
Loop Trip Count	308321	309444	310569	311696	312825	313956	315089	316224
Iteration Latency	3	3	4	3	4	4	3	4
BRAM	2	4	6	8	10	12	14	16
DSP48	9	19	34	56	83	117	154	203
FF	175	339	588	1077	1237	1658	2780	3147
LUT	310	544	937	1381	2010	2585	3212	4672
BRAM %	0.71	1.43	2.14	2.86	3.57	4.29	5.00	5.71
DSP48 %	4.09	8.64	15.45	25.45	37.73	53.18	70.00	92.27
FF %	0.16	0.32	0.55	1.01	1.16	1.56	2.61	2.96
LUT %	0.58	1.02	1.76	2.60	3.78	4.86	6.04	8.78

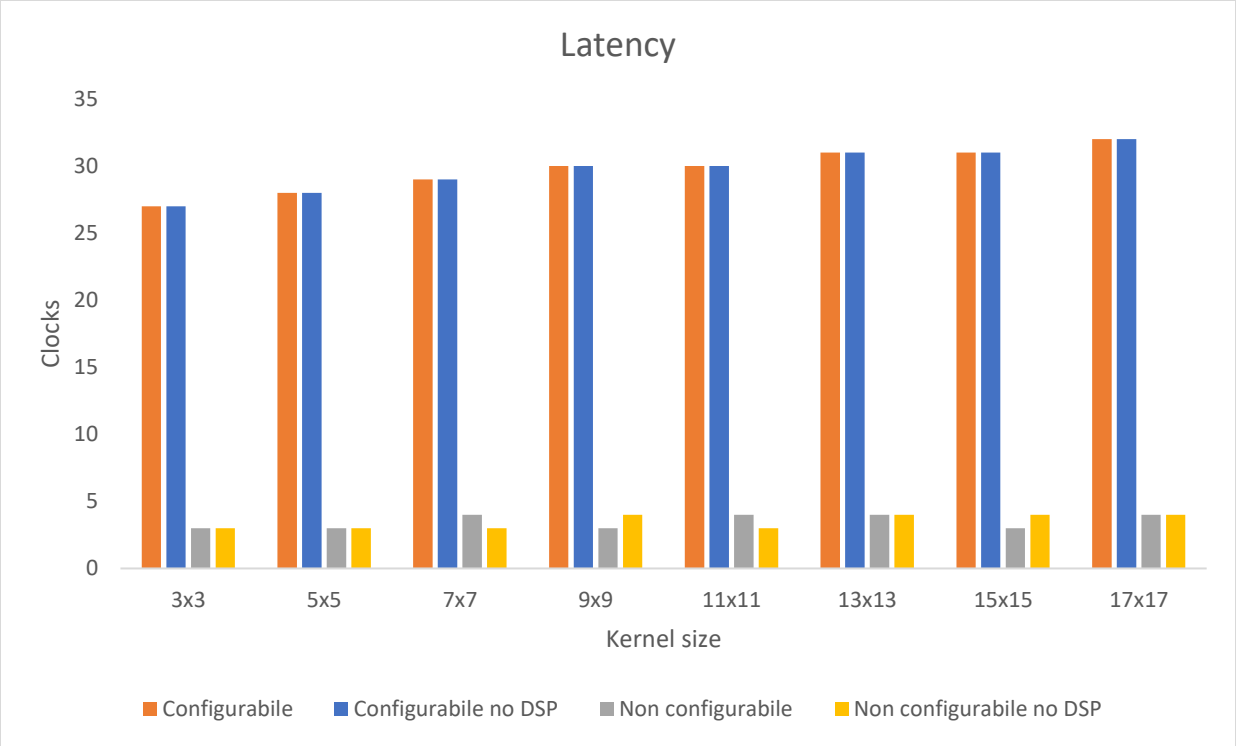
Kernel size	3x3	5x5	7x7	9x9	11x11	13x13	15x15	17x17
Timing	26.75	32.38	34.29	19.93	36.18	22.13	22.73	24.10
Total Latency	308324	309447	310573	311699	312289	313690	315092	316228
Loop Latency	308322	309445	301571	311697	312287	313688	315090	316226
Loop Trip Count	308321	309444	310569	311696	312825	313956	315089	316224
Iteration Latency	3	3	3	4	3	4	4	4
BRAM	2	4	6	8	10	12	14	16
DSP48	1	2	2	2	2	2	3	4
FF	175	339	567	880	1215	1657	2141	3194
LUT	856	2050	3784	6090	9002	12350	16106	21412
BRAM %	0.71	1.43	2.14	2.86	3.57	4.29	5.00	5.71
DSP48 %	0.45	0.91	0.91	0.91	0.91	0.91	1.36	1.82
FF %	0.16	0.32	0.53	0.83	1.14	1.56	2.01	3.00
LUT %	1.61	3.85	7.11	11.45	16.92	23.21	30.27	40.25

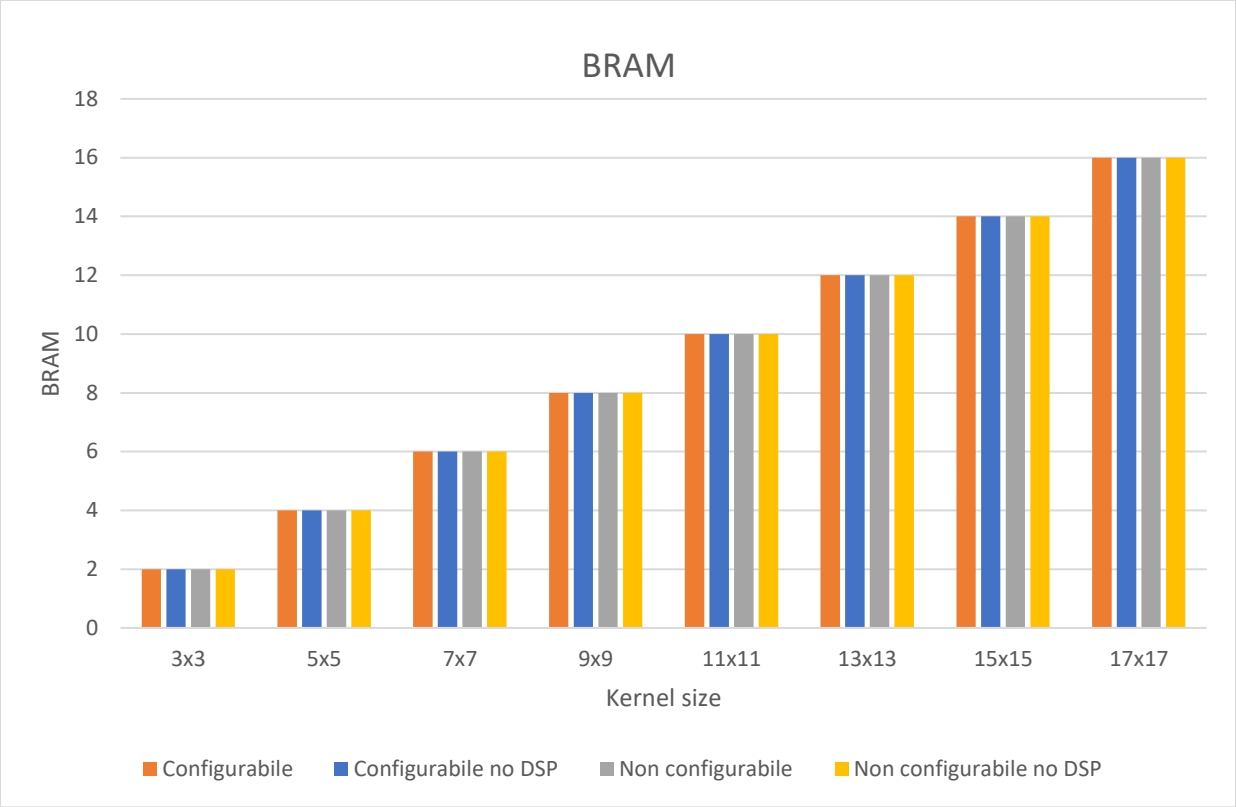
Nei seguenti grafici vengono messe a confronto alcune statistiche delle quattro versioni del filtro.

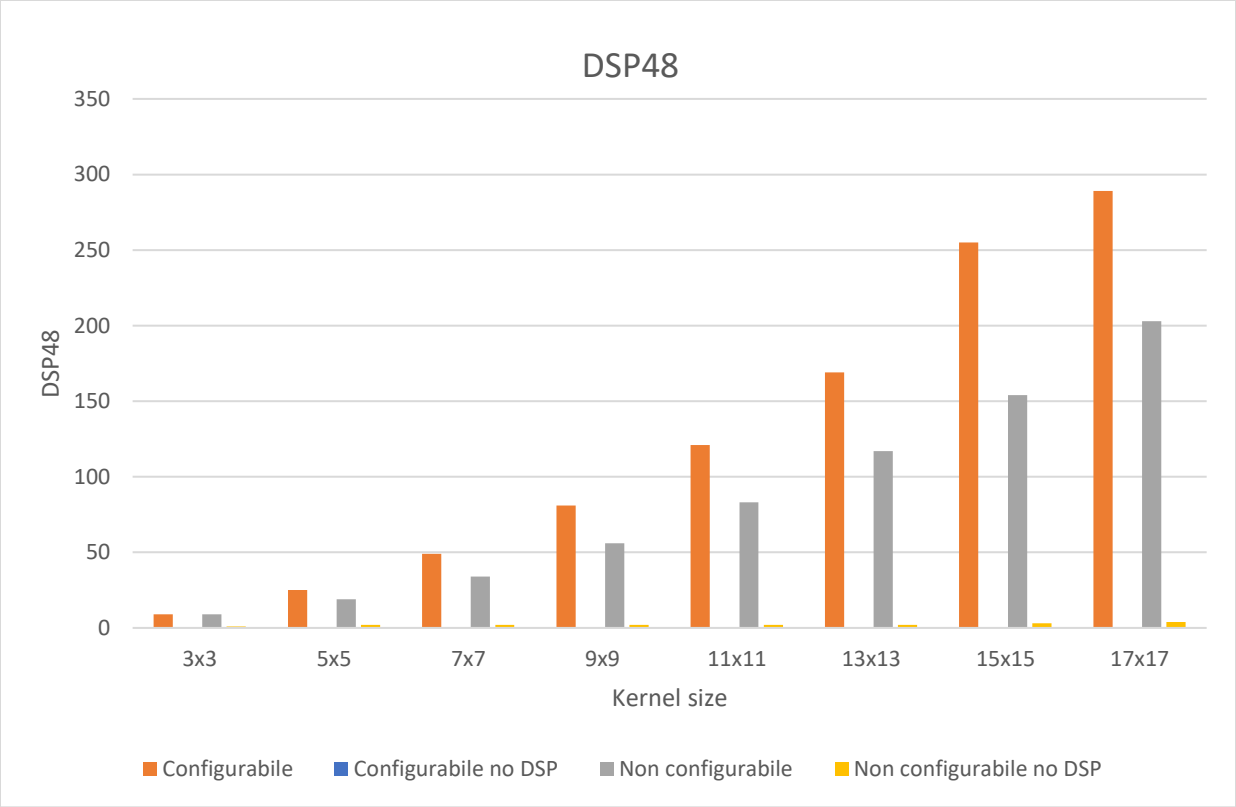


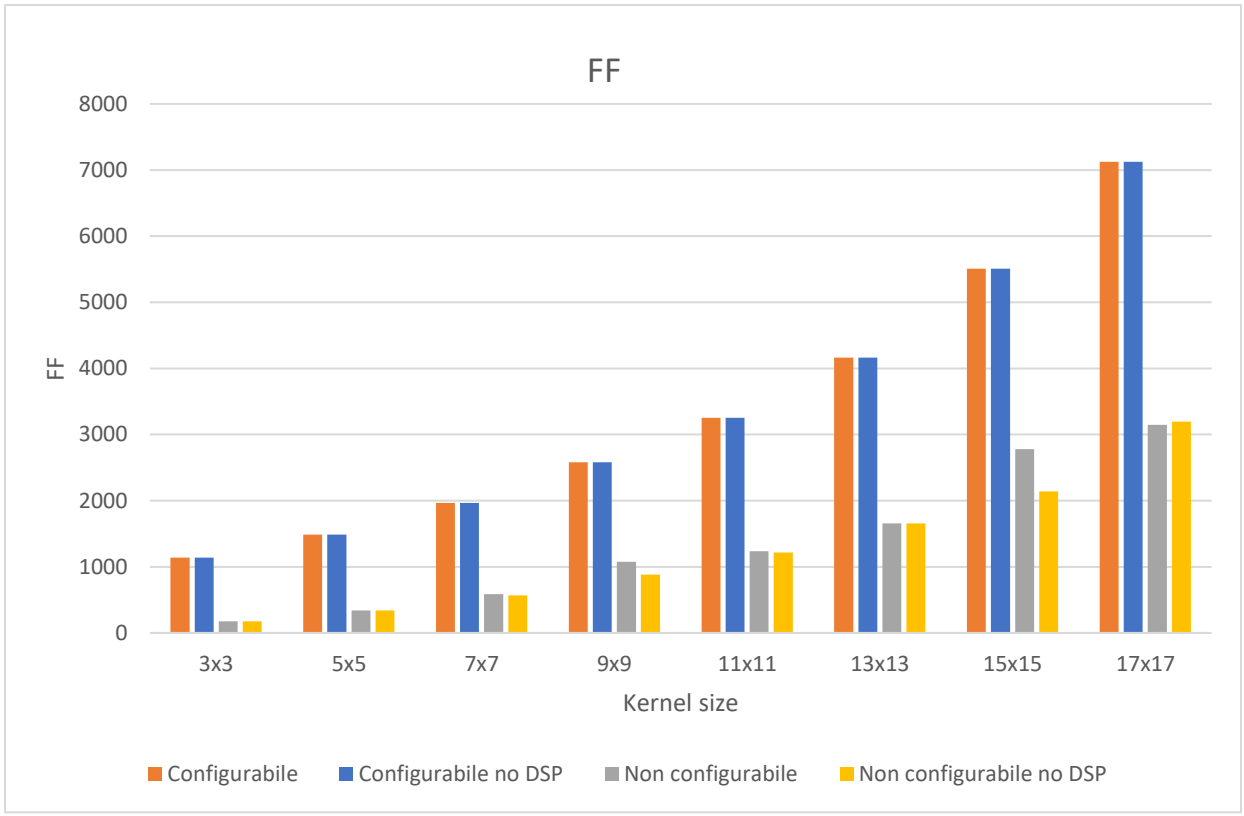
Nel complesso, i tempi di esecuzione rimangono tutti entro il periodo di clock (41.67 ns). Nel caso dei due filtri configurabili, le tempistiche aumentano progressivamente all'aumentare delle dimensioni del kernel. Nella versione senza DSP, l'esecuzione è lievemente più rapida. Per quanto concerne invece i due filtri non configurabili, i tempi oscillano, a causa delle ottimizzazioni effettuate da HLS. È osservabile infatti una correlazione tra la latenza e i tempi dei filtri non configurabili: a maggiore latenza corrisponde infatti un tempo di esecuzione minore.

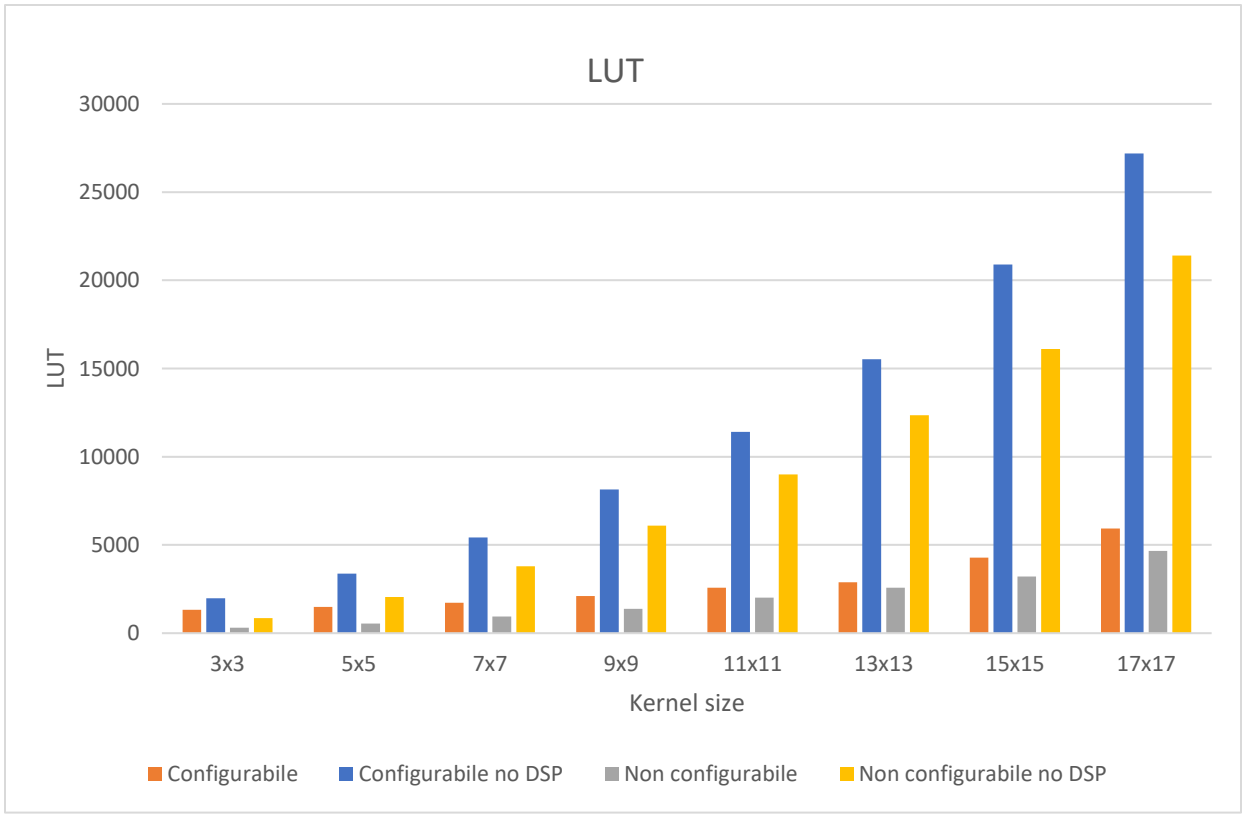
Il grafico "Latency" mostra il numero di step in cui viene suddivisa l'esecuzione in pipeline. A causa delle ottimizzazioni di HLS (conoscendo a tempo di compilazione i pesi del filtro), la latenza viene notevolmente ridotta nelle versioni non configurabili.




BRAM
18
16
14
12
10
8
6
4
2
0
BRAM
3x3
5x5
7x7
9x9
11x11
13x13
15x15
17x17
Kernel size
Configurabile
Configurabile no DSP
Non configurabile
Non configurabile no DSP


DSP48
350
300
250
200
150
100
50
0
DSP48
3x3
5x5
7x7
9x9
11x11
13x13
15x15
17x17
Kernel size
Configurabile
Configurabile no DSP
Non configurabile
Non configurabile no DSP


FF
8000
7000
6000
5000
4000
3000
2000
1000
0
FF
3x3
5x5
7x7
9x9
11x11
13x13
15x15
17x17
Kernel size
Configurabile
Configurabile no DSP
Non configurabile
Non configurabile no DSP


LUT
30000
25000
20000
15000
10000
5000
0
LUT
3x3
5x5
7x7
9x9
11x11
13x13
15x15
17x17
Kernel size
Configurabile
Configurabile no DSP
Non configurabile
Non configurabile no DSP

Circa le risorse utilizzate:

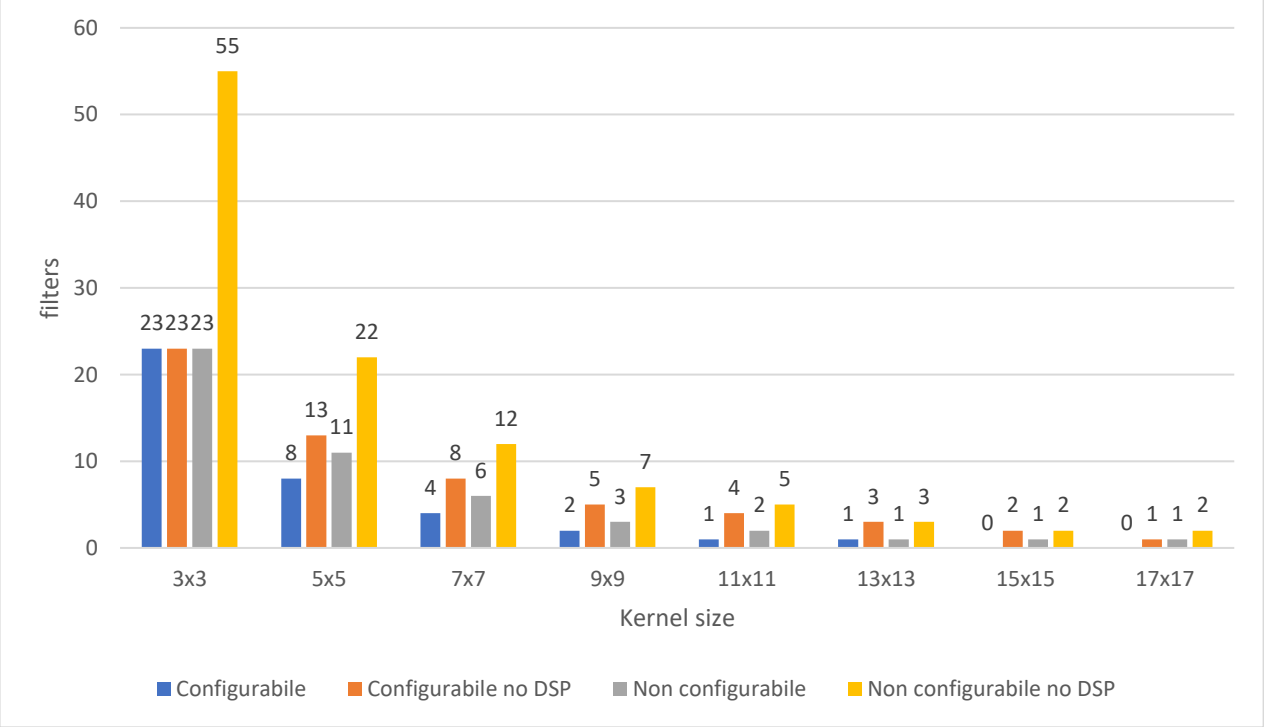
- L’utilizzo di BRAM è identico per tutte le versioni del filtro.
- L’utilizzo di DSP cresce molto rapidamente con l’aumentare delle dimensioni del kernel. Nella versione non configurabile, questa crescita risulta ridotta nella versione non configurabile, a causa probabilmente delle ottimizzazioni operate da HLS. Le due versioni che non utilizzano i DSP per le operazioni MAC hanno un impiego di DSP drasticamente ridotto: nullo per la versione configurabile, prossimo allo zero altrimenti.
- L’utilizzo di FF è identico nelle due versioni configurabili, anche in questo caso vi è una crescita molto rapida in relazione alle dimensioni del kernel. Nelle due versioni non configurabili, l’utilizzo di Flip Flop è drasticamente ridotto, probabilmente a causa di ottimizzazioni da parte di HLS.
- L’utilizzo di LUT risulta superiore in entrambe le versioni configurabili. Esso risulta notevolmente amplificato nelle versioni che limitano l’utilizzo di DSP, mantenendosi comunque entro il 51% nel caso peggiore.

Capitolo 7 - Conclusioni

Grazie ai dati raccolti, è stato possibile effettuare una stima sul numero di filtri (di ciascun tipo) che sarebbe possibile integrare contemporaneamente sulla board, tenendo in conto anche le risorse necessarie a sintetizzare anche i restanti componenti del diagramma a blocchi, riportate nella seguente tabella.

Tipo Risorsa	Quantità
BRAM	6
DSP	8
FF	8627
LUT	6120

Delle quattro versioni realizzate, è stato determinato il tipo di risorsa più utilizzato: DSP per le versioni standard e LUT per le versioni con limitazione sui DSP. È stato calcolato quindi il rapporto tra le risorse disponibili e quelle impiegate dal filtro, approssimando per difetto a 1. I risultati sono messi a confronto nel grafico seguente, seppur con le dovute incertezze relative alle versioni non configurabili.



Com'è osservabile, l'utilizzo di LUT al posto di DSP porta vantaggi in termini di risorse, bilanciando meglio le percentuali di utilizzo, è infatti possibile sintetizzare un numero di filtri maggiore. Nel caso della versione non configurabile, utilizzando le LUT al posto dei DSP è possibile integrare approssimativamente il doppio dei filtri, per la versione configurabile, il rapporto cresce all'aumentare della dimensione del kernel, rendendo persino possibile la sintesi dove non sarebbe stato possibile altrimenti (15x15 e 17x17).

Come era prevedibile, l'introduzione della configurabilità ha un impatto sulle risorse utilizzate e riduce il numero di filtri sintetizzabili. In rapporto con la versione non configurabile, quella configurabile si mantiene lievemente più costosa nella realizzazione di filtri con kernel di dimensione maggiore (dal 13x13 al 17x17), mentre nel caso di kernel più ridotti, (3x3 e 5x5 in particolare) i vantaggi della versione non configurabile in termini di risorse sono notevoli.

Per quanto riguarda il sistema di controllo remoto, l'elevata modularizzazione del codice precedentemente realizzato ha consentito di implementare le dovute estensioni evitando la riscrittura di codice preesistente, limitando le modifiche ai soli moduli che erano stati intesi, fin dal principio, come punti di estensione del progetto (in particolare il modulo "tcp_connection"). L'estensione del progetto ha reso possibile la configurazione del filtro di convoluzione da un host remoto, senza influire significativamente sulle performance. La funzione di configurazione viene infatti invocata solo in caso sia necessario aggiornare la configurazione del filtro, mantenendo un utilizzo di risorse ridotto.

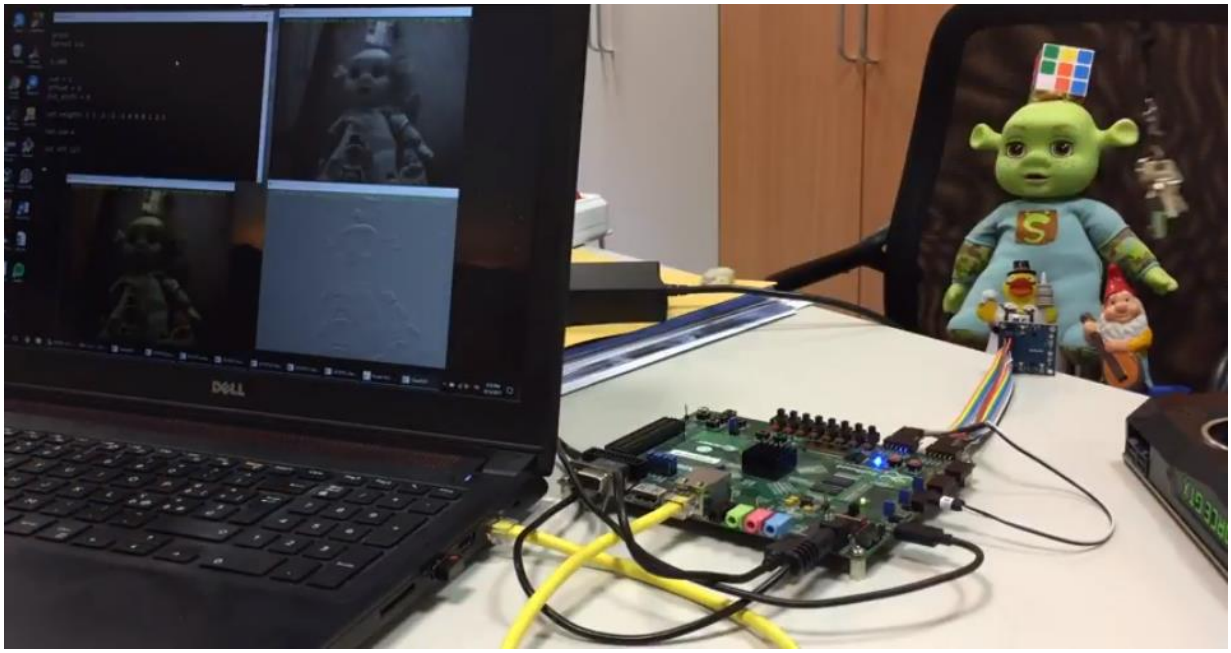


Figura 20: Foto del sistema in esecuzione

La foto mostra l'esecuzione contemporanea del client UDP (Andrea Boscarino) e il client TCP (discusso in questa tesi) su piattaforma Windows. La ZedBoard acquisisce le immagini dal sensore OV7670 ed è collegata al personal computer mediante un cavo ethernet. Gli altri due cavi USB sono necessari rispettivamente per la configurazione della FPGA e per la visualizzazione delle stampe di debug dell'applicazione.

Su schermo vengono mostrate le seguenti schermate:

- Client TCP (in alto a sinistra) da cui viene effettuata la configurazione del filtro
- Stream di immagini in grayscale (in alto a destra)
- Ricostruzione delle immagini a colori RGB (in basso a sinistra)
- Stream di immagini grayscale elaborate (in basso a destra)

Gli altri stream di immagini sono stati nascosti al fine di rendere più comprensibile il contenuto della schermata.

Bibliografia

[1] Mattia Bernasconi – Streaming di immagini via ethernet con sistemi operativi Linux e Baremetal per un sistema di visione embedded con elaborazione su FPGA – Tesi di Laurea, A.A. 2015/2016

[2] Andrea Boscarino – Porting verso Mac e Linux di un client per un sistema di visione embedded – Tesi di Laurea, A.A. 2016/2017

[3] Marco Rossini – Progettazione di filtri di convoluzione separabili su piattaforma embedded – Tesi di Laurea, A.A 2016/2017

[4] http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf

[5] https://www.voti.nl/docs/OV7670.pdf

[6] https://www.xilinx.com/products/design-tools/vivado.html

[7] http://vision.deis.unibo.it/~smatt/Site/Courses.html

[8] http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Improving_Performance.pdf

[9] http://www.cse.usf.edu/~r1k/MachineVisionBook/MachineVision.files/MachineVision_Chapter4.pdf

[10] https://forums.xilinx.com/

[11] http://www.zynqbook.com/

[12] https://savannah.nongnu.org/projects/lwip/

[13] http://www.nongnu.org/lwip/2_0_x/index.html

[14] http://www.codeblocks.org/

[15] https://msdn.microsoft.com/it-it/library/windows/desktop/ms741416.aspx