Alma Mater Studiorum
Università di Bologna
Scuola di Ingegneria e Architettura
Corso di laurea in Ingegneria Informatica

Progettazione di filtri di convoluzione separabili su piattaforma embedded

Tesi di laurea in Calcolatori Elettronici T

Relatore:
Prof. Stefano Mattoccia

Candidato:
Marco Rossini

Correlatori:
Dott. Alessandro Maragno
Dott. Matteo Poggi
Dott. Fabio Tosi

Anno Accademico 2016/2017

Indice

Capitolo 1

Introduzione

La *visione artificiale* (meglio nota in inglese come *computer vision*) è un ramo dell'informatica che studia l'acquisizione, il processamento e l'analisi di caratteristiche presenti in immagini digitali. Essa si propone di riprodurre la vista umana automatizzando la comprensione, l'interpretazione e la valutazione di ciò che è visibile nel mondo reale, allo scopo di emulare, e in certi casi anche di superare, le capacità di riconoscimento proprie di un essere umano. L'estrazione di queste caratteristiche avviene grazie all'utilizzo di *filtri*, ossia algoritmi di *processamento* di un'immagine che attuano su di essa una particolare trasformazione in grado di renderle evidenti.

Per i suoi significativi risvolti pratici, questa scienza riveste oggi un ruolo di primo ordine in numerose applicazioni che costituiscono alcune delle maggiori frontiere della tecnologia moderna, tra le quali: la guida autonoma, il riconoscimento facciale, la ricerca per immagini e la robotica.

Questo lavoro si propone di presentare un'applicazione di visione artificiale nell'ambito della progettazione di una particolare tipologia di filtro che, per le sue proprietà matematiche, viene detto *separabile*. Come si avrà modo di vedere, l'utilizzo (quando possibile) di questa famiglia di filtri costituisce un'ottimizzazione in termini di elaborazione computazionale di un'immagine rispetto all'utilizzo di filtri convenzionali.

Successivamente, verrà presentata un'ulteriore ottimizzazione implementata nella realizzazione del filtro, legata all'utilizzo di un'aritmetica *fixed-point* per la rappresentazione dei numeri da elaborare.

Il filtro realizzato in questo modo sarà destinato ad essere installato su di un circuito integrato programmabile FPGA, più precisamente all'interno dello Zynq-7020 della compagnia Xilinx.

Capitolo 2

Il circuito integrato

La crescita esponenziale di quantità di dati digitali come immagini e video di cui oggi disponiamo ha condotto il mondo informatico alla necessità di poter meglio organizzare ed utilizzare le informazioni in essi contenuti. Uno degli strumenti più potenti in grado di assolvere questo compito è attualmente la *rete neurale convolutiva* (meglio nota in inglese come *convolutional neural network* o *CNN*). Essa è una tipologia di rete neurale artificiale, ispirata all'organizzazione della corteccia visiva animale, oggi largamente impiegata con grande successo nell'elaborazione, classificazione e riconoscimento di immagini.

Poiché le reti neurali richiedono una notevole capacità computazionale, una delle tendenze che ha storicamente prevalso nel loro sviluppo è stata quella di una realizzazione basata su potenti processori grafici (*graphics processing units* o *GPUs*), in contrapposizione a soluzioni, più recenti, basate sui meno onerosi circuiti integrati programmabili (*field programmable gate arrays* o *FPGAs*). La prima scelta comporta l'esecuzione di un *software* che codifica l'algoritmo che descrive la rete, mentre la seconda ne richiede un'implementazione *hardware*. Entrambe le soluzioni presentano vantaggi e svantaggi.

2.1 GPU vs. FPGA

L'implementazione su GPU comporta numerosi vantaggi che le hanno permesso di divenire, nel tempo, lo standard *de facto* per gli algoritmi di visione artificiale. Le GPU mettono infatti a disposizione un grande numero di elementi di processo (*processing elements* o *PE*), un ambiente stabile e in continua espansione, un supporto a infrastrutture standard (come *OpenCL*) e un'ampia disponibilità di proprietà intellettuali per lo sviluppo di applicazio-

ni in maniera rapida. Poiché le GPU eseguono software, inoltre, esse offrono la possibilità di poter riadattare il programma eseguito rapidamente con una semplice modifica del codice, diversamente da soluzioni hardware classiche che non permettono successivi cambiamenti (denominate *hard-core*).

Lo sviluppo dell'industria, tuttavia, ha fatto oggi emergere le FPGA come un valido concorrente delle GPU per l'implementazione di algoritmi di visione artificiale. Esse possono infatti offrire, ad un costo economico molto più contenuto, un'alta affidabilità, un'architettura personalizzabile, una rapida adattabilità (paragonabile a quella delle GPU) grazie alla presenza di una logica riconfigurabile (denominata *soft-core*) e, soprattutto, un consumo energetico notevolmente ridotto.

Un recente studio[4] (2015), condotto da Microsoft, ha messo infatti in evidenza, suscitando un certo interesse da parte del mondo industriale, come l'utilizzo di FPGA possa comportare un risparmio energetico fino a 10 volte superiore rispetto a quello di una GPU per l'esecuzione di uno stesso compito. Il parametro osservato è stato il rapporto tra prestazioni e consumi (throughput/watt) e, sebbene le prestazioni dell'FPGA di riferimento siano risultate molto minori rispetto a quelle della corrispondente GPU, il consumo energetico estremamente ridotto in proporzione si è dimostrato essere un risultato particolarmente consistente, che potrebbe avere conseguenze notevoli per applicazioni in cui le alte prestazioni non costituiscono una priorità.

Un secondo studio[3] ancora più recente (2017), condotto da Intel, ha messo tuttavia in evidenza come in certi casi le FPGA possano addirittura superare le prestazioni di una GPU. Tale studio ha valutato le implementazioni dei più noti algoritmi emergenti di reti neurali in esecuzione tra due generazioni di FPGA (l'Intel Arria 10 e l'Intel Stratix 10) a confronto con uno degli ultimi modelli ad alte prestazioni di GPU (la NVIDIA Titan X Pascal). I risultati hanno dimostrato come, al variare degli algoritmi, le FPGA possano mantenere un consumo energetico dimezzato rispetto alle GPU offrendo prestazioni superiori anche del 60%.

Per queste ragioni, le FPGA rappresentano oggi una realtà molto promettente, che ha catturato per le sue potenzialità l'interesse del mondo della visione artificiale come possibile piattaforma di riferimento per l'implementazione di algoritmi di deep-learning.

Per lo stesso interesse, essa è anche la piattaforma utilizzata in questo progetto, il quale, sebbene a differenza degli esempi citati non abbia come oggetto l'implementazione di una rete neurale nella sua interezza, ne realizza tuttavia un suo modulo, ossia il filtro di convoluzione (con opportune ottimizzazioni).

2.2 Il dispositivo di riferimento: Zynq-7020

Il dispositivo utilizzato in questo progetto è lo Zynq-7020, un *sistema su circuito integrato programmabile* (*all programmable system on chip* o *AP SoC*) appartenente alla famiglia *Zynq-7000* e prodotto dalla compagnia Xilinx. Esso integra un processore hard-core basato su architettura ARM (denominato *processing system*) accoppiato ad una logica riconfigurabile FPGA soft-core (denominata *programmable logic*).

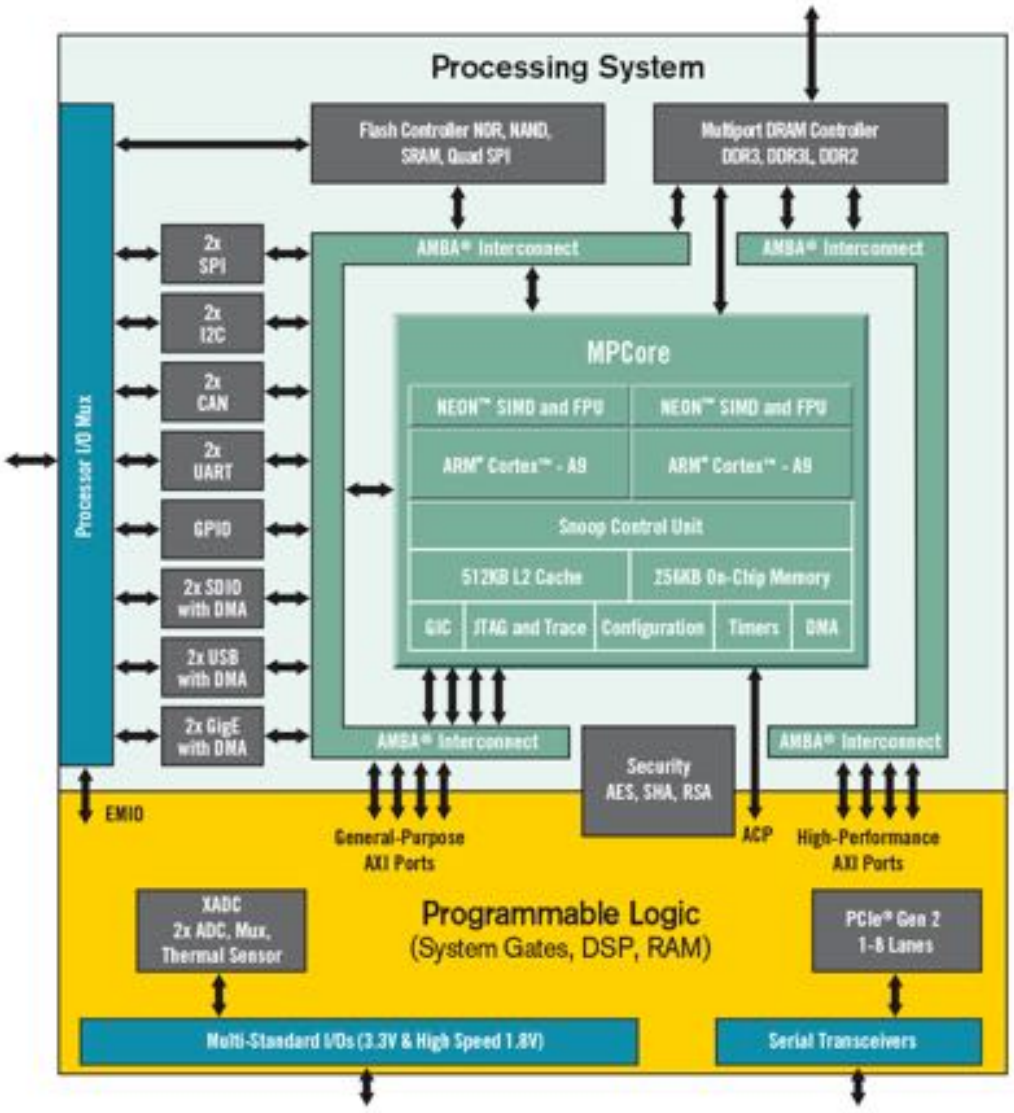

Figura 2.1: Architettura dei dispositivi della famiglia *Zynq-7000* di Xilinx.[7]

Relativamente alla sezione **hard-core**, comune a tutti i dispositivi della famiglia *Zynq-7000*, il dispositivo mette a disposizione:[6]

- 2 CPU ARM Cortex A9 (architettura ARM v7-A);
- cache L1, per data e cache, da 32 KB;
- cache L2, unificata tra i due core, da 512 KB;
- ROM per il boot;
- 256 KB di RAM on chip;
- interfaccia per memorie esterne (DDR2 e DDR3);
- DMA Controller a 8 canali;
- bus AXI (e altri);
- ethernet 10/100/1000 GB;
- USB controller.

Relativamente alla sezione **soft-core**, il dispositivo mette a disposizione:[6]

- 85 000 *celle di logica programmabile* (*programmable logic cells*);
- 140 *block RAMs* (*BRAMs*, in blocchi da 36 KB): tipologia di memorie RAM configurabili incorporate nell'FPGA per la memorizzazione dei dati;
- 220 *digital signal processors* (*DSPs*): microprocessori specializzati nell'esecuzione efficiente di sequenze di istruzioni ricorrenti (come, ad esempio, somme e moltiplicazioni);
- 106 400 *flip flops* (*FFs*): circuiti elementari utilizzati come dispositivi di memoria;
- 53 200 *look-up tables* (*LUTs*): strutture dati che sostituiscono operazioni di calcolo a tempo di esecuzione con più semplici operazioni di consultazione di una tabella.

Come si avrà modo di apprezzare successivamente, l'utilizzo di risorse messe a disposizione dalla sezione soft-core saranno il parametro di valutazione per i risultati sperimentali di questo progetto.

2.3 L'ambiente di sviluppo: Vivado HLS

Lo strumento di sviluppo utilizzato in questo progetto è *Vivado HLS*. Esso è un ambiente di sviluppo integrato (*integrated development environment* o *IDE*), basato su Eclipse, per la programmazione mediante linguaggi di alto livello (C, C++ e SystemC) e l'esportazione dei blocchi logici configurabili (*configurable logic blocks* o *CLB*) presenti nella logica soft-core dell'FPGA.

Le descrizioni delle implementazioni realizzate mediante Vivado HLS saranno fornite nei capitoli successivi, in seguito alla preventiva illustrazione del loro algoritmo.

Capitolo 3

Filtri e separabilità

3.1 Cos'è un filtro

Nell'elaborazione di immagini (*image processing*) il *filtraggio* è un'operazione eseguita su un'immagine allo scopo di migliorarne la qualità o estrarne informazioni di interesse. Il primo utilizzo è, per esempio, di ausilio ai fotografi, per i quali può risultare importante correggere alcuni parametri dell'immagine come luminosità, contrasto e saturazione prima di rendere pubblico uno scatto. Il secondo tipo di utilizzo è invece di interesse della visione artificiale, il cui obiettivo non è la ricerca di un canone estetico, ma la messa in evidenza e la conseguente estrazione di caratteristiche in un'immagine che costituiscono informazione di interesse.

Un *filtro* è un programma che codifica un algoritmo in grado di mettere in atto questa operazione di filtraggio, la quale è ottenuta mediante uno specifico procedimento chiamato *convoluzione*.

Figura 3.1: Schema a blocchi di un filtro (modello *black box*).

3.2 Il processo di convoluzione convenzionale

3.2.1 Significato matematico

La convoluzione (simbolo $\otimes$) è un'operazione matematica che consiste nel processare una matrice attraverso un'altra, detta *matrice di convoluzione* o `kernel`, ottenendo come risultato una nuova matrice. Nel caso bidimensionale, il procedimento prevede di posizionare il kernel nell'angolo in alto a sinistra della matrice da processare, quindi di sommare i prodotti tra ogni elemento del kernel stesso per i corrispondenti elementi con lo stesso indice della matrice di partenza.

Formalmente, date due matrici f e g di uguale dimensione $M \times N$, questa operazione di somma e prodotto, una volta posizionato il kernel, è descritta dalla formula:

$$f \otimes g = \sum_{i=1}^{N} \sum_{j=1}^{M} A_{i,j} \cdot B_{i,j}$$

date quindi due matrici di esempio A e B:

$$A = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}, \; B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

applicando la formula:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \otimes \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} =$$

$$= (a \cdot 1) + (b \cdot 2) + (c \cdot 3) + (d \cdot 4) + (e \cdot 5) + (f \cdot 6) + (g \cdot 7) + (h \cdot 8) + (i \cdot 9)$$

Ciò fornisce il valore della prima cella della matrice risultante. Il procedimento prosegue traslando il kernel verso destra di una quantità costante, detta *stride*, fino al processamento dell'intera matrice di partenza.

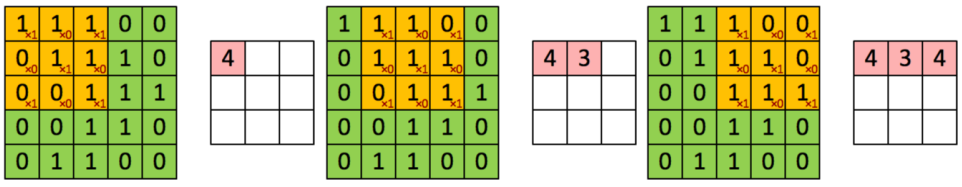

Figura 3.2: Primi tre passi di convoluzione con stride pari a 1 di una matrice 5×5 (in verde) con un kernel 3×3 (in giallo). La matrice risultante ad ogni passo è evidenziata in rosso.

3.2.2 Descrizione dell'algoritmo

L'algoritmo convenzionale di convoluzione di un'immagine è l'analogo del procedimento matematico sopra descritto. In questo caso, la matrice di partenza è l'immagine in ingresso da filtrare, sotto forma di matrice di pixel, e il kernel è un'altra matrice responsabile del filtraggio. La sotto area spazzata dalla traslazione del kernel sull'immagine viene detta *finestra di scorrimento* (o *sliding window*).

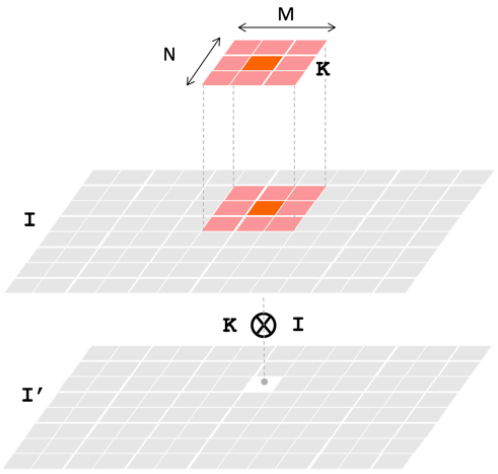

Figura 3.3: Visione tridimensionale della convoluzione tra un **kernel** K e la **finestra di scorrimento** (evidenziata in rosso nello strato I). Il risultato della convoluzione fornisce un valore che, dopo alcune ultime operazioni, costituirà il pixel dell'immagine in uscita I' per la finestra di scorrimento corrente.

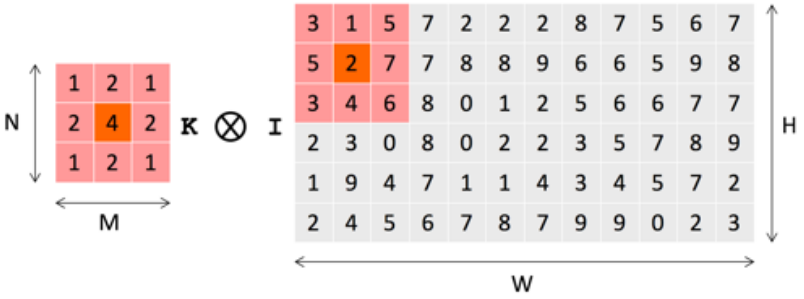

Figura 3.4: Situazione iniziale per la convoluzione di un'immagine con un kernel. A sinistra è rappresentato il kernel K scelto per il filtraggio (sfocatura gaussiana o *gaussian blur*) di dimensione $M \times N$, mentre a destra è rappresentata, sotto forma di matrice di pixel, l'immagine da filtrare I di dimensione $W \times H$. In quest'ultima è evidenziata, in rosso, la posizione iniziale della finestra di scorrimento.

Nel processamento di immagini tuttavia, il valore ottenuto dopo ogni scorrimento non costituirà ancora il valore del pixel dell'immagine in uscita, ma andrà prima diviso per una quantità pari alla somma di tutti i valori del kernel utilizzato (*normalizzazione*). Alcune tipologie di kernel prevedono inoltre la somma per un'ulteriore quantità, detta *offset*. Il risultato di queste operazioni finali fornisce il valore di uno dei pixel che compongono l'immagine filtrata.

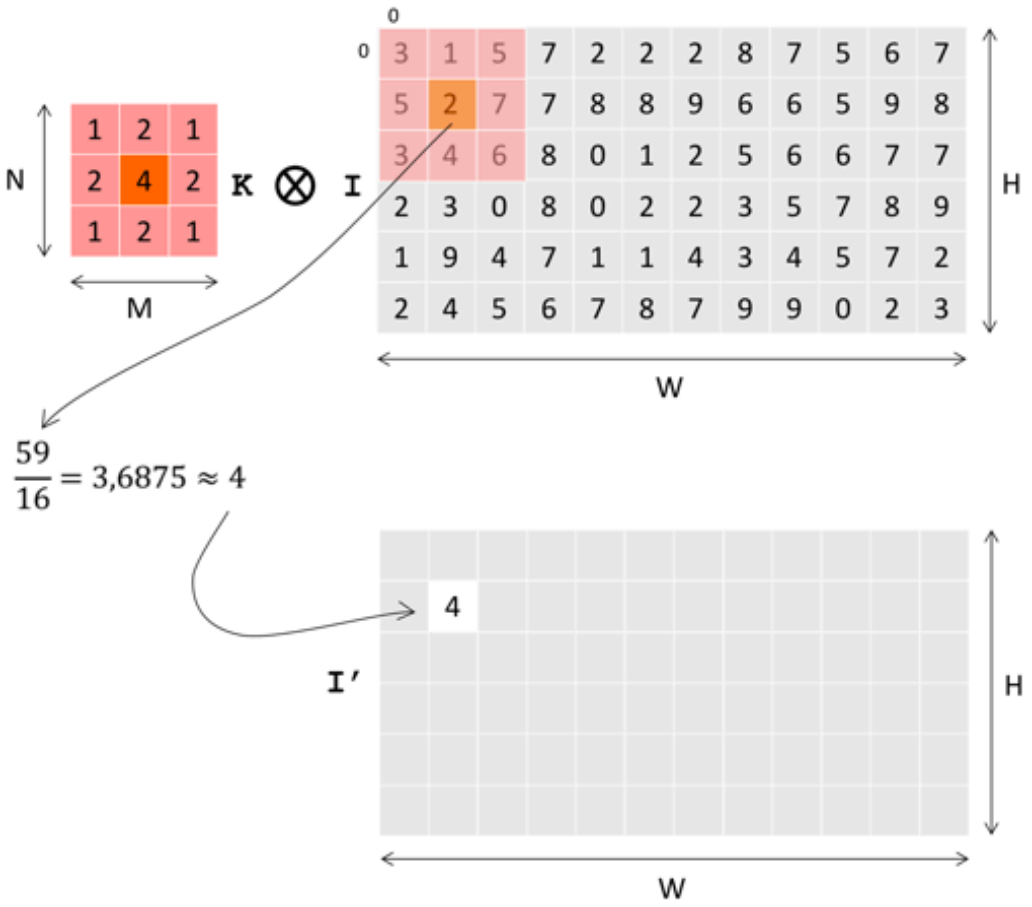

Figura 3.5: Calcolo del valore del primo pixel in uscita (considerando la situazione iniziale di Figura 3.4. Poiché la somma di tutti i valori del kernel utilizzato è 16 (e l'offset è uguale a 0): $I'_{1,1} = \frac{(1\cdot3+2\cdot1+1\cdot5+2\cdot5+4\cdot2+2\cdot7+1\cdot3+2\cdot4+1\cdot6)}{16} = \frac{59}{16} = 3,6875 \approx 4$.

Lo scorrimento della finestra ed i relativi calcoli proseguono fino al processamento dell'intera immagine in input.

3.2.2.1 Note sulle tipologie di kernel

Come osservato, il kernel utilizzato è responsabile della tipologia di filtraggio applicato, e può essere pertanto scelto opportunamente per elaborare l'immagine in un modo piuttosto che in un altro, a seconda degli obiettivi che ci si pone. Un esempio di questi, tra i più rilevanti per scopi di riconoscimento di immagini, è sicuramente l'estrazione dei contorni (*edge detection*). Poiché infatti il contorno di un oggetto costituisce un elemento di separazione tra l'oggetto stesso e ciò che lo circonda, l'estrazione del suo contorno è molto spesso il primo passo per il riconoscimento di ciò che tale oggetto rappresenta.

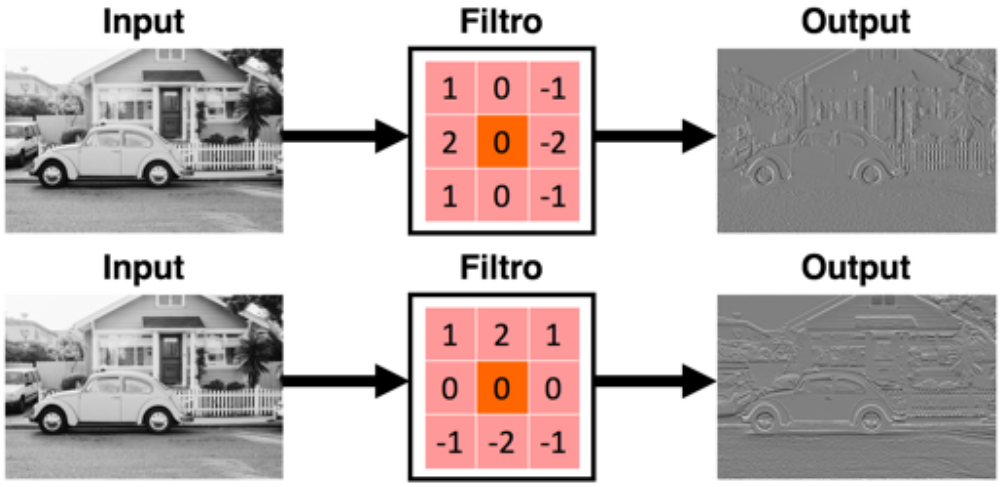

Figura 3.6: Esempi di utilizzo di filtri per edge detection. *In alto*: convoluzione con un kernel di Sobel verticale. *In basso:* convoluzione con un kernel di Sobel orizzontale.

3.2.2.2 Gestione dei bordi

Un problema correlato al processamento di un'immagine mediante convoluzione è quello dei bordi. Il processo di convoluzione porta infatti ad una riduzione delle dimensioni D' dell'immagine processata, in accordo alla formula:

$$D' = (W - M + 1) \cdot (H - N + 1)$$

Dove $W \times H$ è la dimensione dell'immagine in ingresso e $M \times N$ la dimensione del kernel.

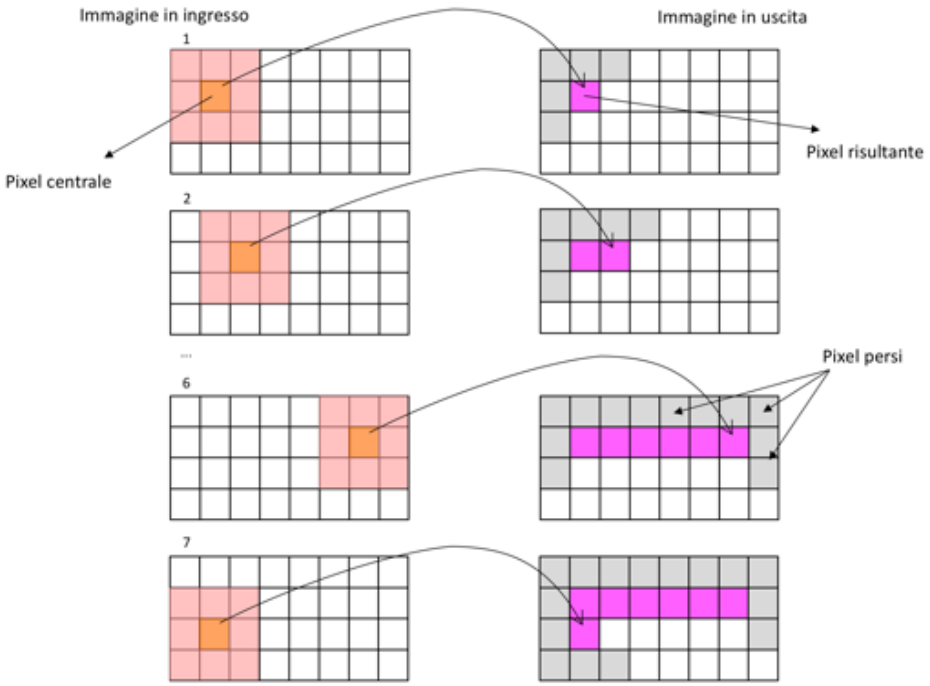

Figura 3.7: Perdita di pixel (in grigio) e conseguente riduzione dell'immagine causati dal processo di convoluzione.

Ciò è concettualmente riconducibile al fatto che un'immagine filtrata in uscita I' ha caratteristiche significative solo in un sottoinsieme dell'immagine in ingresso I. Ignorare il problema porterebbe ad ottenere in uscita un'immagine più piccola, il che potrebbe essere spesso non gradito. Per evitare che ciò accada, è sufficiente aggiungere all'immagine di partenza una cornice di pixel, detta *padding*, di spessore $\frac{M-1}{2} \times \frac{N-1}{2}$. Così facendo, si riconduce il problema alla convoluzione di un'immagine di partenza più grande di quella originale, ottenendo in uscita un'immagine filtrata più piccola, ma delle dimensioni desiderate.

Per la scelta dei valori dei pixel da utilizzare come padding si possono operare scelte differenti, le quali determinano l'aspetto del bordo dell'immagine in uscita. Si può attribuire ad ogni pixel di padding:

1. il valore del pixel più vicino disponibile;
2. un valore costante (es. 0);
3. un valore casuale/non inizializzato.

Per le implementazioni di questo progetto si assumerà sempre l'ultima scelta, poiché risulta la più efficiente in termini computazionali non richiedendo l'allocazione di risorse dedicate.

3.2.3 Complessità computazionale

Un parametro di riferimento per la valutazione dell'efficienza di un algoritmo è il numero di operazioni svolte, le quali ne determinano la sua *complessità computazionale*. Per questo algoritmo, le operazioni elementari necessarie sono elencate in Tabella 3.1.

Operazioni	**Complessità**
Moltiplicazioni	$MN \cdot WH$
Somme	$(MN - 1) \cdot WH$
Divisioni	WH
Totale	$2MN \cdot WH$

Tabella 3.1: Numero di operazioni necessarie all'algoritmo per l'elaborazione, assumendo in ingresso un kernel generico di dimensioni $M \times N$ ed un'immagine generica di dimensioni $W \times H$.

La complessità computazionale cresce quindi in funzione di $M \times N$. Questa quantità sarà il parametro di intervento per le ottimizzazioni di questo progetto.

Negli esempi presentati si è utilizzato, a scopi illustrativi, un kernel di dimensione 3×3. Le dimensioni del kernel possono però variare, determinando il numero di operazioni necessarie a processare l'immagine in ingresso in accordo alla complessità computazionale. Per questo progetto, si è assunto come riferimento l'intervallo di kernel quadrati di dimensioni dispari comprese nell'intervallo $[3 \times 3,\ 17 \times 17]$.[1] Le immagini utilizzate saranno invece sempre di dimensione 640×480.

In Tabella 3.2 sono elencate le operazioni necessarie all'elaborazione per ogni dimensione del kernel nell'intervallo di riferimento. Questi valori verranno ripresi nei paragrafi successivi per scopi di confronto.

[1] formalmente, la dimensione N del kernel deve appartenere all'insieme $\{N \in \mathbb{N};\ 3 \leq N \leq 17, N = 2n + 1, n \in \mathbb{N}\}$.

Kernel	Operazioni
3×3	5 529 600
5×5	15 360 000
7×7	30 105 600
9×9	49 766 400
11×11	74 342 400
13×13	103 833 600
15×15	138 240 000
17×17	177 561 600

Tabella 3.2: Numero di operazioni necessarie all'algoritmo per l'elaborazione di un'immagine 640×480, per ogni dimensione del kernel nell'intervallo di riferimento.

3.2.4 Descrizione dell'implementazione

In Vivado HLS, l'algoritmo appena descritto viene implementato all'interno di una funzione qui denominata `Conventional_Filter`. Tale funzione è organizzata con un doppio ciclo `for` per l'iterazione su tutta la matrice di pixel dell'immagine in ingresso, ottimizzato mediante l'utilizzo della direttiva:

```
#pragma HLS PIPELINE
```

Tale direttiva notifica allo strumento di sintesi di eseguire le operazioni interne in *pipeline*, ossia in una modalità nella quale i passi successivi della sequenza di istruzioni vengono eseguiti a turno in maniera concorrente. In questo modo, l'istruzione successiva ha inizio prima che la precedente sia terminata, determinando una riduzione del numero di cicli di clock necessari a compiere tutte le operazioni. La direttiva riceve inoltre in ingresso il parametro `II=1`, che specifica la volontà di eseguire un'iterazione per clock.

Si riporta di seguito la struttura del codice sorgente appena descritto:

```
#include "ap_int.h"
typedef ap_uint<8> pixel;

void Conventional_Filter(pixel input_img[640*480], pixel
    output_img[640*480])
{
#pragma HLS INTERFACE axis port=out_img
#pragma HLS INTERFACE axis port=in_img

Loop_row: for (int row = 0; row < HEIGHT + (N-1)/2; row++)
  Loop_col: for int col = 0; col < WIDTH + (M-1)/2; col++)
  {
    #pragma HLS PIPELINE II=1

    //codice del filtro
  }
}
```

3.2.4.1 Strutture dati utilizzate

All'interno del doppio ciclo `for` è definita la logica di processamento, nella quale cooperano quattro strutture dati, così definite:

1. `line_buffer`: matrice di dimensioni $(N-1) \times W$ che ha il compito di tenere in memoria le prime $N-1$ righe dell'immagine da elaborare, dove N è l'altezza del kernel e W la larghezza dell'immagine.

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

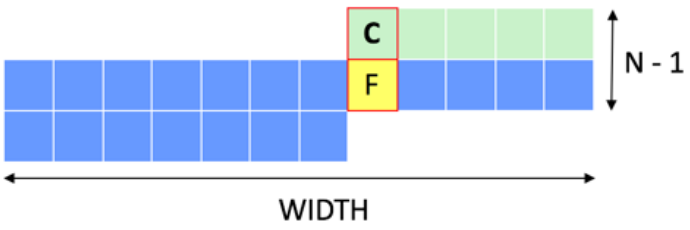

Figura 3.8: Struttura dati `line_buffer`.

2. `window`: matrice di dimensioni $M \times N$ che rappresenta la finestra di scorrimento dell'algoritmo di convoluzione.

```
//sliding window
static pixel window[N][M];
#pragma HLS ARRAY_PARTITION variable=window complete dim=0
```

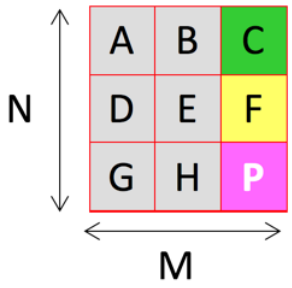

Figura 3.9: Struttura dati `window`.

3. `kernel`: matrice di dimensioni $M \times N$ che rappresenta il kernel di convoluzione scelto per l'operazione di filtraggio.

```
//convolution kernel
static pixel kernel[N][M];
#pragma HLS ARRAY_PARTITION variable=kernel complete dim=0
```

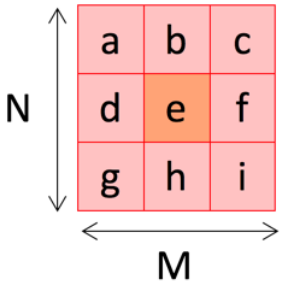

Figura 3.10: Struttura dati `kernel`.

3.2.4.2 Istruzioni

Il meccanismo di cooperazione tra le strutture dati prevede una fase di inizializzazione, necessaria al riempimento di `line_buffer`, a seguito della quale possono avere inizio i prodotti di convoluzione. L'algoritmo è descritto dai seguenti passi:

1. si aggiornano e predispongono le strutture dati `window` e `line_buffer`. La struttura dati `window` subisce uno shift verso sinistra, allo scopo di liberare la sua colonna più a destra al fine di ospitare N nuovi pixel, i quali sono dati dallo scorrimento di `window` verso destra sull'immagine da elaborare. Le prime $N-1$ celle della colonna così liberata vengono riempite con i valori correnti di `line_buffer`. Dopo questa operazione, la struttura dati `line_buffer` subisce uno shift verso l'alto del suo vettore verticale con indice della colonna corrente, al fine di liberare spazio per ospitare il prossimo pixel dell'immagine in ingresso;

```
//shift di window
for (int ii = 0; ii < N; ii++)
for (int jj = 0; jj < M - 1; jj++)
  window[ii][jj] = window[ii][jj+1];

//copia degli N-1 valori da line buffer a window e
    successivo shift di line buffer
if (col < WIDTH)
for (int ii = 0; ii < N - 1; ii++) {
  window[ii][M - 1] = line_buffer[ii][col];

      if (ii < N - 2)
  line_buffer[ii][col] = line_buffer[ii + 1][col];
}
```

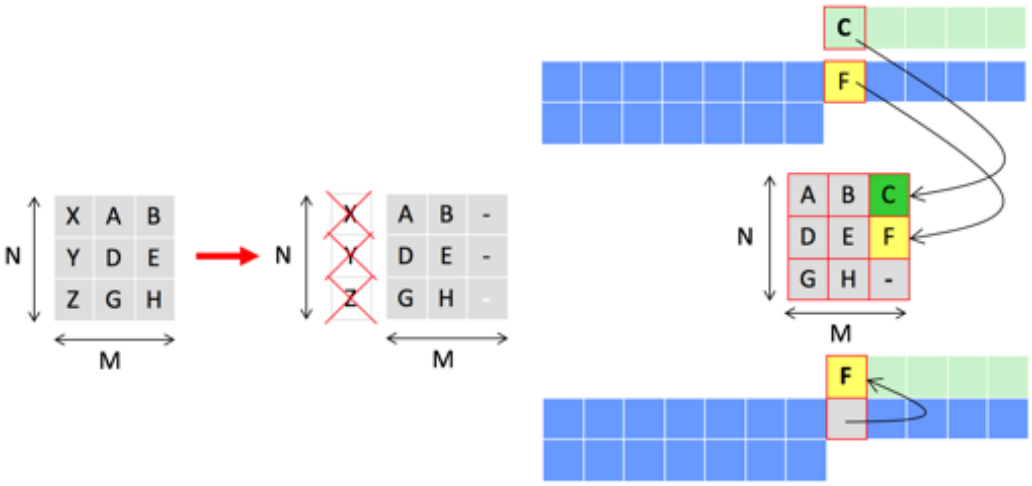

Figura 3.11: *A sinistra:* shift verso sinistra della struttura dati `window`. *A destra:* copia degli $N-1$ valori da `line_buffer` a `window`, quindi shift verso l'alto del vettore verticale di `line_buffer` con indice della colonna corrente.

2. si legge il prossimo pixel P dell'immagine in ingresso. Tale pixel viene memorizzato nell'angolo inferiore destro di `window` liberato precedentemente (al fine di eseguire, successivamente, la convoluzione) e in `line_buffer` (al fine di aggiornare le righe memorizzate).

   Poiché per eseguire la successiva convoluzione sono necessari i valori delle prime due righe dell'immagine di ingresso, il passo 1 e il passo 2 verranno ripetuti $(N-1) \cdot W$ volte senza eseguire il passo 3 (convoluzione), al fine di inizializzare opportunamente la struttura dati `line_buffer`;

```
//lettura di un nuovo pixel, aggiornamento di window e line
    buffer
if (col < WIDTH && row < HEIGHT)
{
  pixel in_temp = in_img[row * WIDTH + col];
  window[N - 1][KERNEL_WIDTH - 1] = in_temp;
  line_buffer[N - 2][col] = in_temp;
}
```

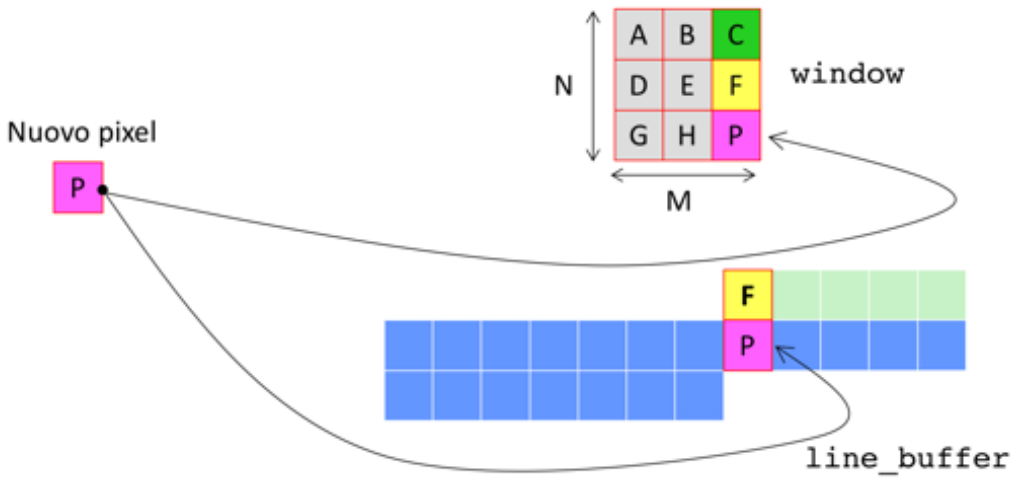

Figura 3.12: Meccanismo di lettura di un nuovo pixel e sua memorizzazione in `window` e `line_buffer`.

3. ora che la struttura dati `line_buffer` è correttamente inizializzata, si esegue la convoluzione di `window` con `kernel`, ottenendo in uscita un valore che verrà diviso per la somma di tutti i valori di `kernel` (e a cui verrà sommato l'eventuale offset). Ciò costituisce il pixel dell'immagine di uscita per la finestra corrente. Nel caso in cui `window` contenga valori non inizializzati, si avrà sull'immagine in uscita un effetto di bordo di un'entità proporzionale al padding utilizzato.

   La funzione che incapsula il prodotto di convoluzione, la normalizzazione e l'aggiunta dell'offset è `pixel_weighted_average`;

```
//calcolo del pixel in uscita
if (row >= (N-1)/2 && col >= (M-1)/2)
{
      pixel out = pixel_weighted_average(kernel, kern_sum,
   kern_off, window);
      out_img[(row - (N-1)/2) * WIDTH + (col - (N-1)/2)] =
   out;
}

//prodotto di convoluzione, normalizzazione e aggiunta di
   eventuale offset
pixel pixel_weighted_average(s_int kernel[N][M], s_int
   kern_sum, s_int kern_off, pixel window[N][N])
{
#pragma HLS INLINE

  ap_int<MAC_BITS> out_temp = 0;

  pixel result = 0;

  //prodotto di convoluzione
  Edge_i:  for (int i = 0; i < N; i++)
    Edge_j: for (int j = 0; j < M; j++)
      out_temp = out_temp + window[i][j] * kernel[i][j];

  result = out_temp / kern_sum + kern_off;

  return (result)(7,0);
}
```

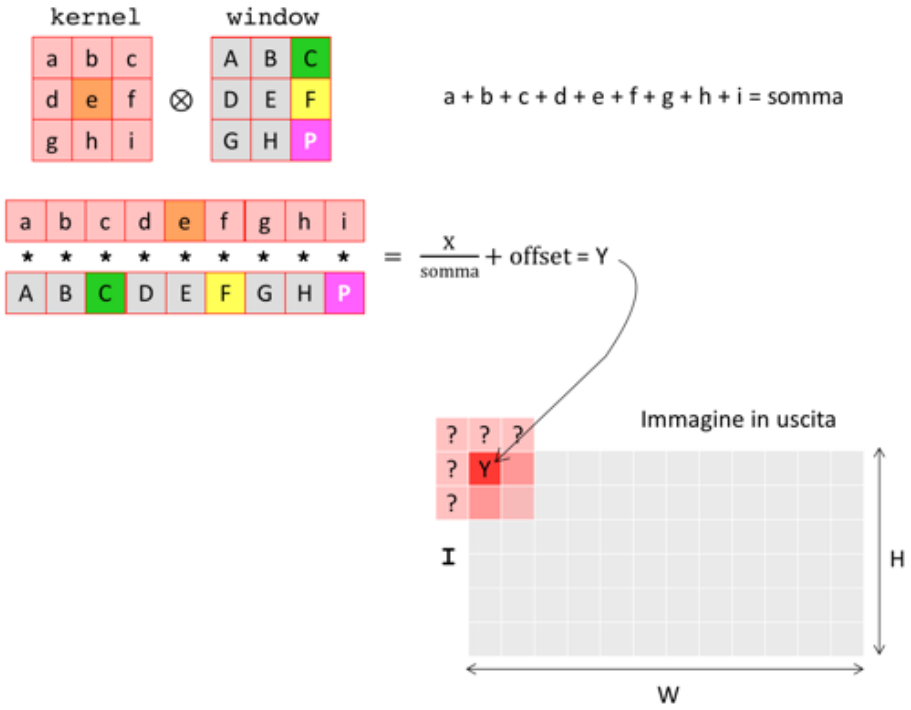

Figura 3.13: Prodotto di convoluzione, normalizzazione, aggiunta di eventuale offset e aggiornamento dell'output.

4. si riprende quindi dal passo 1 fino al termine del numero di pixel dell'immagine in ingresso.
   Raggiunto tale punto si otterrà in uscita l'immagine filtrata.

L'implementazione appena illustrata rappresenta il bersaglio di ottimizzazione di questo lavoro. Come si vedrà nel paragrafo successivo, l'utilizzo di una particolare famiglia di kernel, quando utilizzabile, permetterà all'algoritmo di essere significativamente più efficiente.

3.3 Il processo di convoluzione separabile

3.3.1 Separabilità di un filtro

Come precedentemente accennato, esistono sottoinsiemi di kernel di convoluzione che godono di specifiche proprietà: uno tra questi è quello dato dai kernel *separabili*. Un filtro K viene detto separabile se può essere ottenuto dal *prodotto esterno* (simbolo $\otimes$) di un vettore colonna a con un vettore riga b:

$$K = a \otimes b$$

considerando un kernel K 3×3 generico di esempio, deve quindi valere:

$$\begin{bmatrix} a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_1 & a_3b_2 & a_3b_3 \end{bmatrix} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \otimes \begin{bmatrix} b_1 & b_2 & b_3 \end{bmatrix}$$

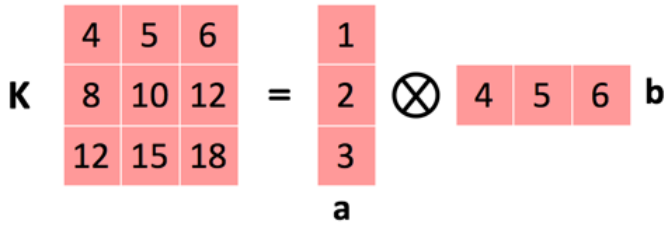

Figura 3.14: Scomposizione di un kernel separabile K nei suoi vettori componenti a e b.

Se ciò è verificato, il kernel risulta perfettamente separabile nei suoi componenti a e b. Sfortunatamente però, in termini pratici, solo una piccola frazione dei kernel di convoluzione utilizzabili risultano essere perfettamente separabili, e ciò avviene per i kernel di rango 1. Esistono tuttavia procedimenti matematici in grado di trasformare un filtro non separabile in uno separabile, a meno di un'approssimazione (tra questi, la *singular value decomposition* o *SVD*).

Questo tipo di scomposizione rappresenta la chiave per la prima ottimizzazione presentata in questo progetto, poiché permette, grazie ad un utilizzo opportuno, la scrittura di un algoritmo in grado di risparmiare una notevole quantità di operazioni (e quindi di risorse) rispetto all'algoritmo convenzionale. Questa nuova soluzione rappresenta tuttavia, nei casi in cui il kernel non

sia perfettamente separabile, un'approssimazione del precedentemente metodo, pertanto i risultati ottenuti non risulteranno ad esso sempre equivalenti (mentre lo saranno nel caso in cui il kernel utilizzato sia perfettamente separabile). Adottando strategie opportune tuttavia, è possibile rendere accettabile l'approssimazione ottenuta, permettendo l'utilizzo di un'implementazione semplice ed efficace da parte di sistemi embedded come le FPGA.

3.3.2 Descrizione dell'algoritmo

L'algoritmo separabile di convoluzione di un'immagine si discosta operativamente da quello convenzionale. Se l'algoritmo convenzionale consiste infatti in una convoluzione a due dimensioni (ovvero tra due matrici bidimensionali), l'algoritmo separabile consiste invece in due convoluzioni ad una dimensione. La garanzia di equivalenza tra i due procedimenti è fornita dalla proprietà associativa della convoluzione.

Considerando infatti un kernel separabile K, i suoi vettori componenti a e b ed una matrice da processare M, vale:

$$K \otimes M = a \otimes (b \otimes M) = b \otimes (a \otimes M)$$

Sfruttando questa proprietà, è quindi possibile riscrivere l'algoritmo di convoluzione di un'immagine, articolandolo nei seguenti passi:

1. si esegue la convoluzione tra il vettore verticale componente del kernel per una finestra verticale di dimensioni equivalenti, posizionata sull'angolo in alto a sinistra dell'immagine da processare. Il risultato così ottenuto viene memorizzato su una struttura dati d'appoggio;

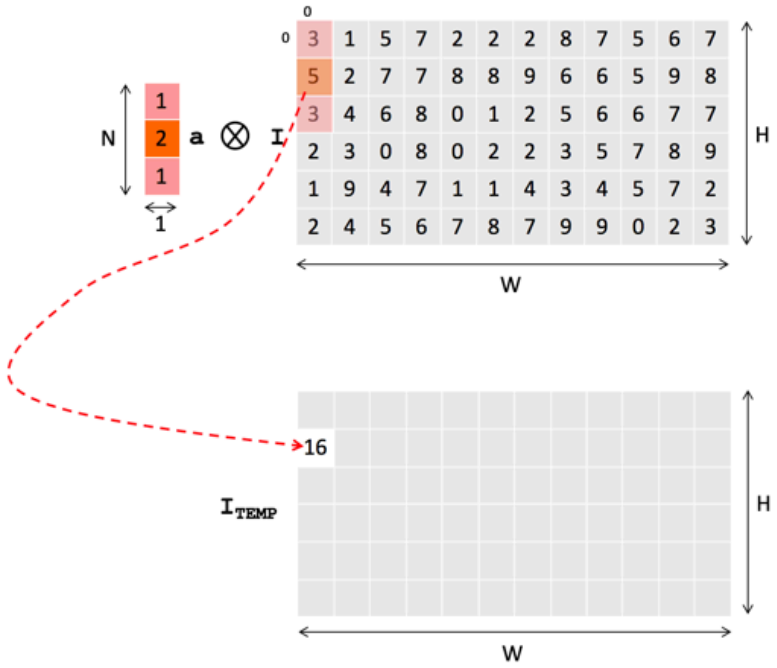

Figura 3.15: Illustrazione del passo 1 dell'algoritmo per un kernel 3×3. Il risultato del prodotto di convoluzione tra il componente verticale del kernel a e la finestra di dimensioni equivalenti posta sull'immagine I viene memorizzata su una struttura dati di appoggio I_{temp}.

2. si ripete il passo 1 dell'algoritmo $N - 1$ volte, dove N è la dimensione del kernel, scorrendo la finestra verso destra;

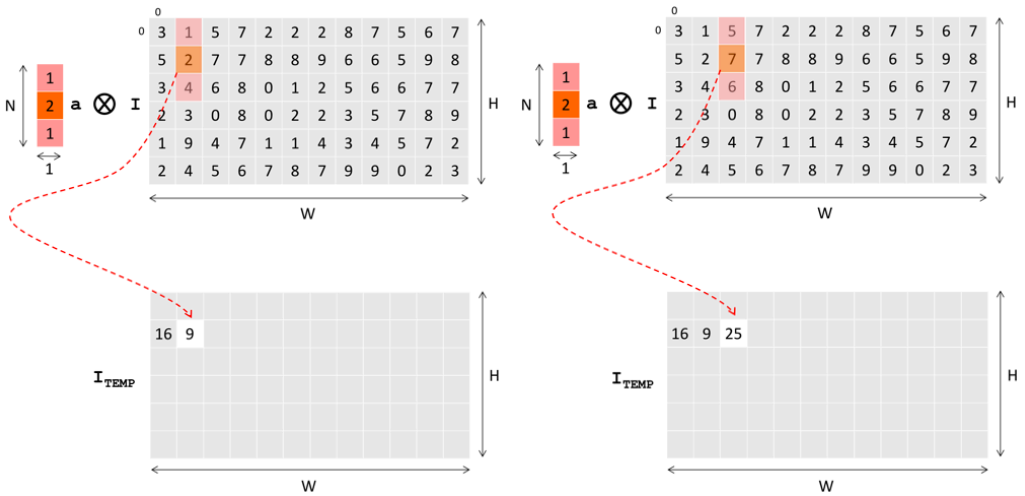

Figura 3.16: Ripetizione del passo 1 dell'algoritmo $N - 1$ volte.

3. si esegue la convoluzione tra il vettore orizzontale componente del kernel per una finestra orizzontale di dimensioni equivalenti, posizionata sugli N risultati intermedi precedentemente ottenuti. Si ottiene così il valore da normalizzare e a cui sommare un eventuale offset. Il valore risultante è il pixel dell'immagine filtrata in uscita;

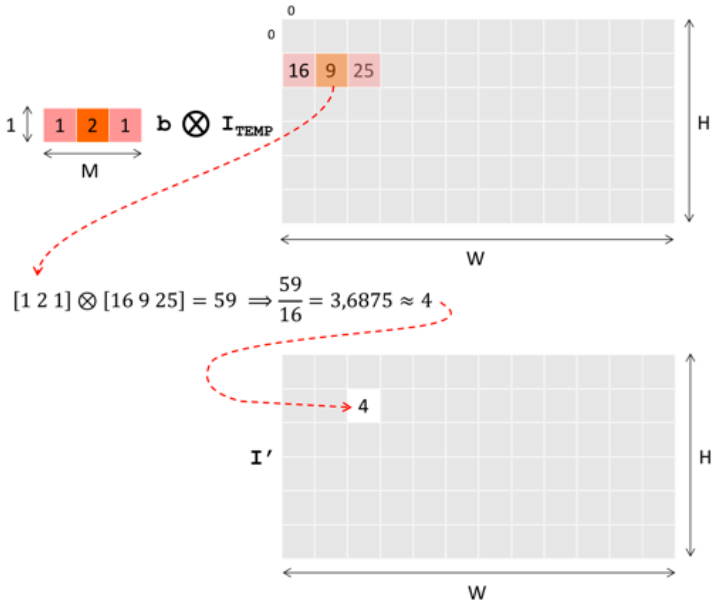

Figura 3.17: Illustrazione del passo 3 dell'algoritmo.

4. si ripete il passo 1, traslando verso destra la finestra di scorrimento, ed il passo 3 fino al processamento dell'intera riga corrente in ingresso. Non è infatti più necessario eseguire il passo 2, poiché gli $N-1$ risultati parziali necessari sono già disponibili in quanto precedentemente calcolati. Al termine della riga corrente, poiché è necessario reinizializzare `conv_buffer`, si riprende nuovamente ad eseguire anche il passo 2.

Come si vedrà in Sezione 3.3.4 tuttavia, l'implementazione si discosterà leggermente dall'algoritmo presentato, in accordo alle scelte sulla gestione dei bordi specificate in Sezione 3.2.2.2. Non si avrà infatti una fase di inizializzazione della struttura dati deputata alla memorizzazione dei risultati intermedi, ma si eseguiranno fin da subito i passi 1 e 3 senza eseguire il passo 2. Nella sezione che segue si terrà pertanto conto della complessità computazionale dell'implementazione e non dell'algoritmo (pur discostandosi da questa di quantità trascurabili).

3.3.3 Complessità computazionale

Per questo algoritmo, le operazioni elementari necessarie a confronto con l'algoritmo convenzionale sono elencate in Tabella 3.3.

Operazioni	**Complessità convenzionale**	**Complessità separabile**
Moltiplicazioni	$MN \cdot WH$	$(M+N) \cdot WH$
Somme	$(MN-1) \cdot WH$	$2(N-1) \cdot WH$
Divisioni	WH	WH
Totale	$2MN \cdot WH$	$(M+3N-1) \cdot WH$
Differenza	$(2MN - M - 3N + 1) \cdot WH$	

Tabella 3.3: Complessità dell'algoritmo di convoluzione convenzionale (maggiore) a confronto con quella dell'algoritmo di convoluzione separabile (minore).

Le operazioni necessarie per le applicazioni di questo progetto sono invece elencate in Tabella 3.4, a confronto con quelle già viste per il metodo convenzionale.

Kernel	**Operazioni convenzionale**	**Operazioni separabile**	**Differenza**
3×3	5 529 600	3 379 200	2 150 400
5×5	15 360 000	5 836 800	9 523 200
7×7	30 105 600	8 294 400	21 811 200
9×9	49 766 400	10 752 000	39 014 400
11×11	74 342 400	13 209 600	61 132 800
13×13	103 833 600	15 667 200	88 166 400
15×15	138 240 000	18 124 800	120 115 200
17×17	177 561 600	20 582 400	156 979 200

Tabella 3.4: Numero di operazioni necessarie all'elaborazione di un'immagine 640×480 per l'algoritmo di convoluzione separabile a confronto con l'algoritmo di convoluzione convenzionale. Nella colonna *Differenza* è possibile visualizzare il numero di operazioni risparmiate.

3.3.4 Descrizione dell'implementazione

In Vivado HLS, l'algoritmo appena descritto viene implementato all'interno di una funzione qui denominata `Separable_Filter`. L'organizzazione di tale funzione è analoga a quella vista per l'algoritmo convenzionale.

Se ne riporta di seguito la struttura nel codice sorgente:

```
#include "ap_int.h"
typedef ap_uint<8> pixel;

void Separable_Filter(pixel input_img[640*480], pixel output_img
    [640*480])
{
#pragma HLS INTERFACE axis port=out_img
#pragma HLS INTERFACE axis port=in_img

Loop_row: for (int row = 0; row < HEIGHT + (N-1)/2; row++)
  Loop_col: for int col = 0; col < WIDTH + (M-1)/2; col++)
  {
    #pragma HLS PIPELINE II=1

    //codice del filtro
  }
}
```

3.3.4.1 Strutture dati utilizzate

Le strutture dati allocate per questo algoritmo sono le stesse viste in Sezione 3.2.4.1, con l'eccezione di `window`, la quale non è più una matrice di dimensione $M \times N$, ma un vettore di lunghezza N.

```
//sliding window
static pixel window[N];
#pragma HLS ARRAY_PARTITION variable=window complete dim=0
```

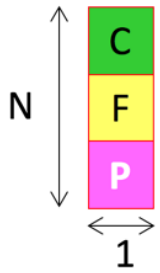

Figura 3.18: Struttura dati `window` per l'implementazione separabile.

Viene inoltre introdotta una struttura dati aggiuntiva di appoggio rispetto all'implementazione convenzionale, al fine di memorizzare i risultati intermedi di ogni passo convolutivo. Essa è un vettore di interi dimensione $N-1$. Tale struttura dati, qui denominata `conv_buffer`, rappresenta il costo aggiuntivo in termini allocazione di memoria rispetto all'implementazione convenzionale. Come si avrà modo di apprezzare meglio nelle sezioni successive, tuttavia, ciò costituisce un compromesso estremamente vantaggioso in termini di utilizzo di risorse complessivo.

```
//buffer di convoluzione
static pixel conv_buffer[N-1];
#pragma HLS ARRAY_PARTITION variable=conv_buffer complete dim=0
```

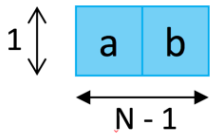

Figura 3.19: Struttura dati `conv_buffer`.

3.3.4.2 Istruzioni

Il meccanismo di cooperazione tra le strutture dati prevede, come per l'algoritmo convenzionale, la fase di inizializzazione necessaria al riempimento di `line_buffer`, a seguito della quale possono avere inizio i prodotti di convoluzione separabili. L'algoritmo è descritto dai seguenti passi:

1. si aggiornano e predispongono le strutture dati `window` e `line_buffer`. Le prime $N-1$ celle della struttura dati `window` vengono riempite con i valori correnti di `line_buffer`, mentre la struttura dati `line_buffer` subisce uno shift verso l'alto del suo vettore verticale con indice della colonna corrente, al fine di liberare spazio per il prossimo pixel dell'immagine in ingresso;

```
//copia degli N-1 valori da line buffer a window e shift di
    line buffer
for(int i = 0; i < N - 1; i++)
{
  window[i] = line_buffer[i][col];
  if (i < N - 2)
    line_buffer[i][col] = line_buffer[i + 1][col];
}
```

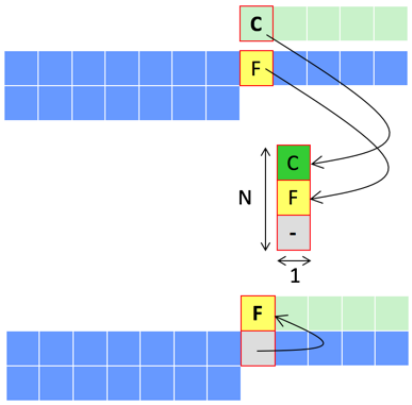

Figura 3.20: Copia degli $N-1$ valori dalla struttura dati `line_buffer` a `window` e shift di `line_buffer`.

2. si legge il prossimo pixel P dell'immagine in ingresso. Tale pixel viene memorizzato nell'ultima cella in basso di `window`, non riempita precedentemente, (al fine di eseguire, successivamente, la convoluzione) e in `line_buffer` (al fine di aggiornare le righe memorizzate).

   Poiché per eseguire la successiva convoluzione sono necessari i valori delle prime due righe dell'immagine di ingresso, il passo 1 e il passo 2 verranno ripetuti $(N-1) \cdot W$ volte senza eseguire il passo 3 (convoluzione), al fine di inizializzare opportunamente la struttura dati `line_buffer`;

```
//lettura di un nuovo pixel, aggiornamento di window e line
    buffer
if(col < WIDTH && row < HEIGHT)
{
  pixel in_temp = in_img[row * IMG_WIDTH + col];
  window[N - 1] = in_temp;
  line_buffer[N - 2][col] = in_temp;
}
```

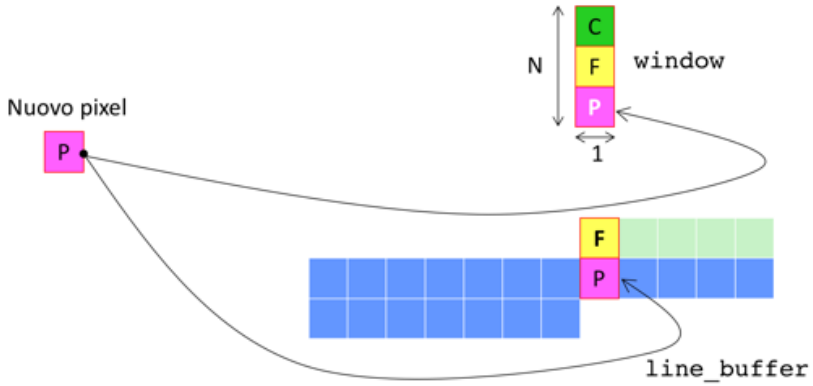

Figura 3.21: Meccanismo di lettura di un nuovo pixel e sua memorizzazione in `window` e `line_buffer`.

3. ora che la struttura dati `line_buffer` è correttamente inizializzata, si esegue la convoluzione di `window` con la prima colonna di `kernel`.[2] Questa operazione fornisce il valore intermedio con il quale si può eseguire la convoluzione tra `conv_buffer` ed il componente vettore verticale del kernel separabile. Si ottiene quindi il valore che verrà diviso per la somma di tutti i valori di `kernel` (e a cui verrà sommato l'eventuale offset), il quale costituisce il valore del pixel in output. Contemporaneamente alla seconda convoluzione, viene eseguito uno shift verso sinistra di `conv_buffer`, al fine di liberare spazio per la memorizzazione del risultato intermedio della prima convoluzione. Nel caso in cui `window` contenga valori non ancora inizializzati, si avrà sull'immagine in uscita un effetto di bordo di un'entità proporzionale al padding utilizzato. La funzione che incapsula il prodotti di convoluzione, la normalizzazione e l'aggiunta dell'offset è `pixel_weighted_average_separable`;

[2]per gli utilizzi pratici di questo filtro, si è assunto che i componenti vettore verticale e vettore orizzontale del kernel separabile coincidano sempre, rispettivamente, con la prima colonna e la prima riga del kernel stesso. In generale, tuttavia, ciò non è sempre matematicamente verificato. Possono infatti esistere kernel separabili i cui componenti verticale e orizzontale non siano uguali, rispettivamente, alla loro prima colonna e prima riga.

```
//calcolo del pixel in uscita
if (row >= (N-1)/2 && col >= (M-1)/2)
{
      pixel out = pixel_weighted_average(kernel, kern_sum,
   kern_off, window);
      out_img[(row - (N-1)/2) * WIDTH + (col - (N-1)/2)] =
   out;
}

//prodotti di convoluzione, normalizzazione e aggiunta di
   eventuale offset
pixel pixel_weighted_average_separable
(s_int kernel_h[N], s_int kernel_v[N], s_int kern_sum,
   s_int kern_off, pixel window[N])
{
#pragma HLS INLINE

  static ap_int<CONV_BUFFER_BITS> convolution_buffer[N];

  ap_int<CONV_BUFFER_BITS> temp_v = 0;
  ap_int<RESULT_BITS> temp_h = 0;

  pixel result = 0;

  vertical_convolution:  for(int i = 0; i < N; i++)
    temp_v = temp_v + kernel_v[i] * window[i];

  horizontal_shift: for(int j = 0; j < N-1; j++)
  {
    convolution_buffer[j] = convolution_buffer[j + 1];
    temp_h = temp_h + convolution_buffer[j]*kernel_h[j];
  }

  temp_h = temp_h + temp_v * kernel_h[N - 1];
  convolution_buffer[N - 1] = temp_v;

  result = temp_h / kern_sum + kern_off;

  return (result)(PIXEL_BITS - 1,0);
}
```

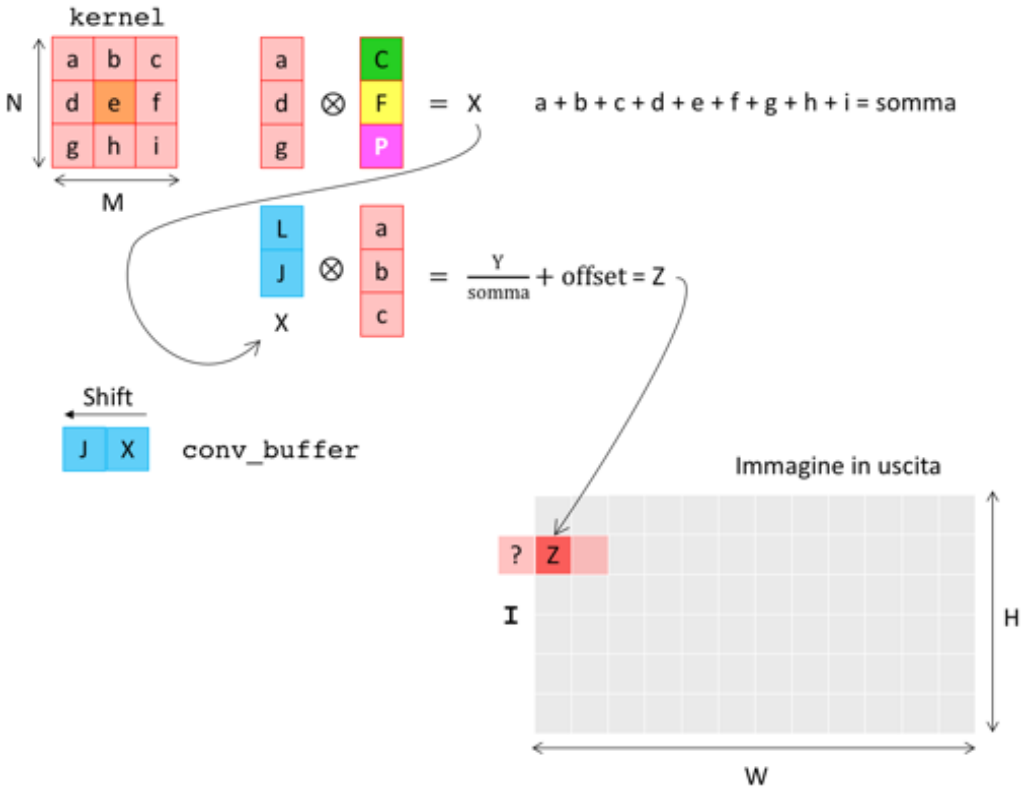

Figura 3.22: Prodotti di convoluzione e shift di `conv_buffer`, normalizzazione, aggiunta di eventuale offset e aggiornamento dell'output.

4. si riprende quindi dal passo 1 fino al termine del numero di pixel dell'immagine in ingresso.
   Raggiunto tale punto si otterrà in uscita l'immagine filtrata.

Capitolo 4

L'aritmetica fixed-point

La metodologia di rappresentazione dei numeri è un aspetto che assume in informatica un ruolo cruciale, poiché determina il carico di elaborazione dei processori deputati all'esecuzione delle loro operazioni matematiche (nel nostro caso, i *DSPs*). I numeri reali possono essere rappresentati secondo due modalità: in aritmetica *fixed-point* (o *a virgola fissa*) e in aritmetica *floating-point* (o *a virgola mobile*).

4.1 Fixed-point vs. floating-point

4.1.1 Fixed-point

La rappresentazione mediante aritmetica fixed-point prevede che il numero reale che si vuole rappresentare venga mantenuto in memoria sotto forma di numero intero. Tale intero è ottenuto moltiplicando il numero in ingresso per un valore pari ad una potenza di due, detto *fattore di scala*, quindi troncando la parte decimale del valore così ottenuto (tramite un *cast* ad `int`):

$$\text{fixed point} = \lfloor \text{input} \cdot \text{fattore di scala} \rfloor$$

dove:

$$\text{fattore di scala} = 2^{\text{bits}}$$

Il valore intero risultante da queste operazioni corrisponde alla rappresentazione fixed-point del numero in ingresso. Per riottenere il numero originale, a meno di un'approssimazione, sarà sufficiente dividere l'intero fixed-point per il fattore di scala. L'entità dell'approssimazione è determinata dal numero di bit dedicati alla parte decimale e alla parte intera della rappresentazione: i primi sono pari all'esponente del fattore di scala, mentre i secondi sono pari

alla differenza tra i bit dedicati al tipo di dato intero scelto per eseguire il cast e l'esponente del fattore di scala:

$$\text{bit parte decimale} = \log_2(\text{fattore di scala}) = \text{bits}$$

$$\text{bit parte intera} = \text{bit tipo di dato} - \text{fattore di scala}$$

La posizione della virgola sarà quindi fissa, perché stabilita a priori da queste quantità, e verrà memorizzata internamente dal programma che elabora i numeri così rappresentati durante tutta la sua esecuzione. Il vantaggio principale dell'utilizzo di un'aritmetica fixed-point per la rappresentazione dei dati è dato dall'efficienza della loro elaborazione, poiché la memorizzazione di un valore intero in memoria permette ad un processore di sfruttare le numerose ottimizzazioni di cui dispone per l'elaborazione di questo tipo di dato. Tuttavia, l'utilizzo di questa aritmetica impone anche un limite significativo all'accuratezza dei numeri rappresentabili, poiché stabilisce un vincolo sulla quantità di bit dedicati alla parte intera e alla parte decimale. Questa rappresentazione risulta pertanto ottimale in situazioni in cui è richiesta una precisione assoluta, ma può risultare non ottimale in situazioni che richiedono una precisione relativa specifica per i differenti valori elaborati. In quest'ultimo caso, si può comunque scegliere di mantenere questa rappresentazione intera (con i vantaggi che essa comporta) modulando opportunamente il numero di bit assegnati alla parte intera e decimale, così da soddisfare anche requisiti di precisione elevati. In alternativa, può risultare ottimale una rappresentazione basata su un'aritmetica floating-point.

4.1.2 Floating-point

Un numero rappresentato in aritmetica floating-point non ha infatti un numero specifico di bit dedicati alla parte intera o alla parte decimale. Diversamente, esso riserva un certo numero di bit per un valore, detto *mantissa*, e un certo numero di bit per un *esponente*, che mantiene in memoria la posizione della virgola. Analogamente al metodo di notazione scientifica, il valore in aritmetica floating-point è dato dal prodotto tra la mantissa e il numero 2 elevato all'esponente:

$$\text{floating point} = \text{mantissa} \cdot 2^{\text{esponente}}$$

In questa aritmetica quindi, la rappresentazione del numero porta con sé l'informazione sulla posizione della virgola, la quale non ha pertanto bisogno di essere stabilita a priori, ma rappresenta un'informazione contenuta in ogni numero stesso. Ciò comporta che la virgola possa muoversi lungo tutto il numero, permettendo di rappresentare intervalli di valori anche milioni di volte

più piccoli di quelli offerti da una rappresentazione fixed-point, garantendo un'estrema precisione. Tuttavia, nonostante questa aritmetica metta a disposizione una potente flessibilità, essa risulta molto onerosa a livello computazionale, poiché richiede l'esecuzione di operazioni complesse e dispendiose. Per questa ragione, essa risulta una scelta conveniente per applicazioni in cui l'alta precisione costituisce una priorità, mentre può risultare un collo di bottiglia per sistemi che mettono a disposizione un hardware con risorse limitate. Gli utilizzi di questo lavoro rientrano in quest'ultima casistica, pertanto la rappresentazione dei numeri adottata sarà di tipo fixed-point.

4.2 L'ottimizzazione mediante fixed-point

Come si è potuto osservare nelle descrizioni delle implementazioni presentate in Sezione 3.2.4 e in Sezione 3.3.4, il tipo di dato utilizzato per l'elaborazione dei valori dei pixel delle immagini e dei kernel di convoluzione è sempre `pixel`, ossia un intero (ridefinito tramite `#define`) di default a 8 bit. Tuttavia, i kernel di convoluzione utilizzabili non sono sempre composti da numeri interi, ma sono anzi, molto più spesso, anche composti da valori reali. Ci si potrebbe pertanto domandare come l'utilizzo di un tipo di dato intero sia compatibile con l'elaborazione di valori in ingresso di tipo reale: ciò è possibile grazie all'utilizzo della rappresentazione fixed-point. Come accennato nel paragrafo precedente, la piattaforma utilizzata in questo progetto mette a disposizione un numero di risorse limitato, che risultano pertanto incompatibili con l'utilizzo di un'aritmetica floating-point richiesta per una perfetta rappresentazione di tali valori. La soluzione adottata è stata quindi di utilizzare una rappresentazione in aritmetica fixed-point, sacrificando una parte di precisione al fine di permettere l'elaborazione di tali valori reali sul dispositivo di riferimento.[1] I valori rappresentati internamente quindi, in accordo con quanto precedentemente descritto, risulteranno essere un'approssimazione dei valori originali, comportando una resa non perfetta dell'immagine in uscita. Tuttavia, come si vedrà negli esempi seguenti, si farà in modo che tale disallineamento risulti trascurabile.

Assumendo un kernel K 3×3 di esempio ed una finestra F 3×3 di esempio:

$$K = \begin{bmatrix} 0.13 & -7.61 & 12.10 \\ -0.33 & 0.07 & 5.82 \\ 1.43 & -9.49 & 3.43 \end{bmatrix}, \; F = \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}$$

[1] costituirebbe infatti una forte limitazione permettere l'elaborazione di soli kernel composti da numeri interi.

il loro prodotto di convoluzione $K \otimes F$ è esattamente:

$$K \otimes F = 33,2959$$

Analizziamo ora il valore ottenuto convolvendo la finestra in ingresso con la rappresentazione fixed-point del kernel utilizzato, confrontandolo con il risultato originale al variare dei bit dedicati alla rappresentazione delle cifre decimali del kernel (ovvero l'esponente del fattore di scala):

1 bit (fattore di scala = 2^1):

$$\text{round}(K \times 2^1) = \begin{bmatrix} 0 & -15 & 24 \\ -1 & 0 & 12 \\ 3 & -19 & 7 \end{bmatrix}$$

$$\frac{\begin{bmatrix} 0 & -15 & 24 \\ -1 & 0 & 12 \\ 3 & -19 & 7 \end{bmatrix} \otimes \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}}{2^1} = 32,5$$

$$\Delta = 33,2959 - 32,5 = 0,7959$$

2 bit (fattore di scala = 2^2):

$$\text{round}(K \times 2^2) = \begin{bmatrix} 1 & -30 & 48 \\ -1 & 0 & 23 \\ 6 & -38 & 14 \end{bmatrix}$$

$$\frac{\begin{bmatrix} 1 & -30 & 48 \\ -1 & 0 & 23 \\ 6 & -38 & 14 \end{bmatrix} \otimes \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}}{2^2} = 33,5$$

$$\Delta = 33,2959 - 33,5 = -0,2041$$

3 bit (fattore di scala = 2^3):

$$\text{round}(K \times 2^3) = \begin{bmatrix} 1 & -61 & 97 \\ -3 & 1 & 47 \\ 11 & -76 & 27 \end{bmatrix}$$

$$\frac{\begin{bmatrix} 1 & -61 & 97 \\ -3 & 1 & 47 \\ 11 & -76 & 27 \end{bmatrix} \otimes \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}}{2^3} = 33,5$$

$$\Delta = 33,2959 - 33,5 = -0,2041$$

4 bit (fattore di scala = 2^4):

$$\text{round}(K \times 2^4) = \begin{bmatrix} 2 & -122 & 194 \\ -5 & 1 & 93 \\ 23 & -152 & 55 \end{bmatrix}$$

$$\frac{\begin{bmatrix} 2 & -122 & 194 \\ -5 & 1 & 93 \\ 23 & -152 & 55 \end{bmatrix} \otimes \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}}{2^4} = 33,3125$$

$$\Delta = 33,2959 - 33,3125 = -0,0166$$

5 bit (fattore di scala = 2^5):

$$\text{round}(K \times 2^5) = \begin{bmatrix} 4 & -244 & 387 \\ -11 & 2 & 186 \\ 46 & -304 & 110 \end{bmatrix}$$

$$\frac{\begin{bmatrix} 4 & -244 & 387 \\ -11 & 2 & 186 \\ 46 & -304 & 110 \end{bmatrix} \otimes \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}}{2^5} = 33,0312$$

$$\Delta = 33,2959 - 33,0312 = 0,2647$$

6 bit (fattore di scala = 2^6):

$$\text{round}(K \times 2^6) = \begin{bmatrix} 8 & -487 & 774 \\ -21 & 4 & 372 \\ 92 & -607 & 220 \end{bmatrix}$$

$$\frac{\begin{bmatrix} 8 & -487 & 774 \\ -21 & 4 & 372 \\ 92 & -607 & 220 \end{bmatrix} \otimes \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}}{2^6} = 33,2344$$

$$\Delta = 33,2959 - 33,2344 = 0,0615$$

7 bit (fattore di scala = 2^7):

$$\text{round}(K \times 2^7) = \begin{bmatrix} 17 & -974 & 1549 \\ -42 & 9 & 745 \\ 183 & -1215 & 439 \end{bmatrix}$$

$$\frac{\begin{bmatrix} 17 & -974 & 1549 \\ -42 & 9 & 745 \\ 183 & -1215 & 439 \end{bmatrix} \otimes \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}}{2^7} = 33,3203$$

$$\Delta = 33,2959 - 33,3203 = -0,0244$$

8 bit (fattore di scala = 2^8):

$$\text{round}(K \times 2^8) = \begin{bmatrix} 33 & -1948 & 3098 \\ -84 & 18 & 1490 \\ 366 & -2429 & 878 \end{bmatrix}$$

$$\frac{\begin{bmatrix} 33 & -1948 & 3098 \\ -84 & 18 & 1490 \\ 366 & -2429 & 878 \end{bmatrix} \otimes \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}}{2^8} = 33,3242$$

$$\Delta = 33,2959 - 33,3242 = -0,0283$$

9 bit (fattore di scala = 2^9):

$$\text{round}(K \times 2^9) = \begin{bmatrix} 67 & -3896 & 6195 \\ -169 & 36 & 2980 \\ 732 & -4859 & 1756 \end{bmatrix}$$

$$\frac{\begin{bmatrix} 67 & -3896 & 6195 \\ -169 & 36 & 2980 \\ 732 & -4859 & 1756 \end{bmatrix} \otimes \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}}{2^9} = 33,3047$$

$$\Delta = 33,2959 - 33,3047 = -0,0088$$

10 bit (fattore di scala = 2^{10}):

$$\text{round}(K \times 2^{10}) = \begin{bmatrix} 133 & -7793 & 12390 \\ -338 & 72 & 5960 \\ 1464 & -9718 & 3512 \end{bmatrix}$$

$$\frac{\begin{bmatrix} 133 & -7793 & 12390 \\ -338 & 72 & 5960 \\ 1464 & -9718 & 3512 \end{bmatrix} \otimes \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 4 & 9 & 2 \end{bmatrix}}{2^{10}} = 33,2959$$

$$\Delta = 33,2959 - 33,2959 = 0$$

Gli esempi[2] mostrano come un aumento dei bit dedicati alla rappresentazione della parte decimale comporti, in accordo con quanto precedentemente descritto, una diminuzione dell'errore di approssimazione Δ applicato ai valori originari, fino a renderlo nullo con l'utilizzo di 10 bit. Questi risultati motivano pertanto la scelta di utilizzo di un'aritmetica fixed-point per gli scopi di questo progetto, poiché mostrano come l'approssimazione di tale rappresentazione risulti trascurabile con l'allocazione di una quantità di bit opportuna.

4.2.1 Descrizione dell'implementazione

A livello implementativo, le operazioni di moltiplicazione per il fattore di scala e di troncamento (per la trasformazione in rappresentazione fixed-point) e la successiva divisione per il fattore di scala (per il riottenimento del valore in forma originaria a meno di un'approssimazione) sono presenti in uno strato di logica più esterno al filtro di convoluzione realizzato. Il filtro è infatti inizialmente un programma scritto in linguaggio di alto livello, ma che è successivamente destinato ad essere esportato come blocco logico programmato sul dispositivo FPGA, costituendo pertanto un elemento più interno all'architettura complessiva che pilota l'intero processo di elaborazione di immagini. Il filtro è stato perciò reso configurabile al fine di poter comunicare correttamente con le interfacce (*AXI stream*) deputate al trasferimento dei dati delle immagini in ingresso e uscita dalla telecamera collegata allo Zynq-7020. Lo strato di logica più esterno che gestisce l'interazione con un utente (il quale inserisce, tramite un terminale, i kernel di convoluzione da applicare) è stato infatti realizzato in [1] sull'ambiente di design integrato (*integrated design environment*) SDK. In tale strato, sono presenti le istruzioni che realizzano le operazioni di conversione dei valori del kernel originali inseriti alla loro rappresentazione fixed-point, e viceversa, seppur con una lieve differenza rispetto agli esempi presentati. L'implementazione realizzata non prevede infatti che i valori inseriti dall'utente vengano moltiplicati per il fattore di scala e troncati (in ingresso al filtro) e infine divisi (all'interno del filtro, prima di restituirli), bensì che vengano moltiplicati e troncati sia il kernel inserito dall'utente, sia la somma di tutti i valori di tale kernel, senza eseguire la divisione successivamente. Il risultato è equivalente, poiché all'interno del filtro si dividerà il prodotto di convoluzione tra ogni finestra e il kernel per un valore più grande.[2] Ciò permette di mostrare all'utente i pixel del-

[2] dividere il quoziente di una divisione per un valore è infatti lo stesso che aumentarne il divisore fin dall'inizio.

l'immagine elaborata mediante il kernel scelto a meno di un'approssimazione.

Si presentano di seguito le righe di codice, tratte da [1], che realizzano la conversione in fixed-point del kernel inserito dall'utente (`weights`) e della somma dei valori di tale kernel (`sum`):

```
{
  //conversione in fixed-point del kernel
  temp = config->weights[i * config->width + j] * (1 << config->
   bit_shift);

  //codice di invio dei dati inseriti al filtro, previa verifica
}

{
  //conversione in fixed-point della somma dei valori del kernel
  temp = config->sum * (1 << config->bit_shift);

  //codice di invio dei dati inseriti al filtro, previa verifica
}
```

Capitolo 5

Risultati sperimentali

In questo capitolo vengono presentati i risultati sperimentali ottenuti in questo lavoro, mostrando un confronto tra le risorse impiegate dal filtro di convoluzione convenzionale, ovvero il punto di partenza, e quelle impiegate dal filtro di convoluzione separabile, ovvero l'obiettivo di questo lavoro, realizzato secondo le ottimizzazioni presentate nei capitoli precedenti.

I valori che verranno mostrati sono stati ottenuti tramite lo strumento di sintesi del blocco logico messo a disposizione da Vivado HLS, il quale ne simula il funzionamento fornendo una stima, abbastanza precisa, del suo effettivo utilizzo di risorse.

5.1 Il punto di partenza

Le prestazioni e le risorse utilizzate dall'algoritmo di convoluzione convenzionale, nell'implementazione descritta in Sezione 3.2.4, sono mostrate in Figura 5.1, Figura 5.2, Figura 5.3, Figura 5.4, Figura 5.5 e Figura 5.6.

Per la comprensione delle voci utilizzate si forniscono, in aggiunta alle definizioni già fornite in Sezione 2.2, le seguenti definizioni:[5]

- *timing*: massima frequenza di clock raggiungibile;
- *iteration latency*: numero di cicli di clock necessari per completare un'iterazione di un ciclo.

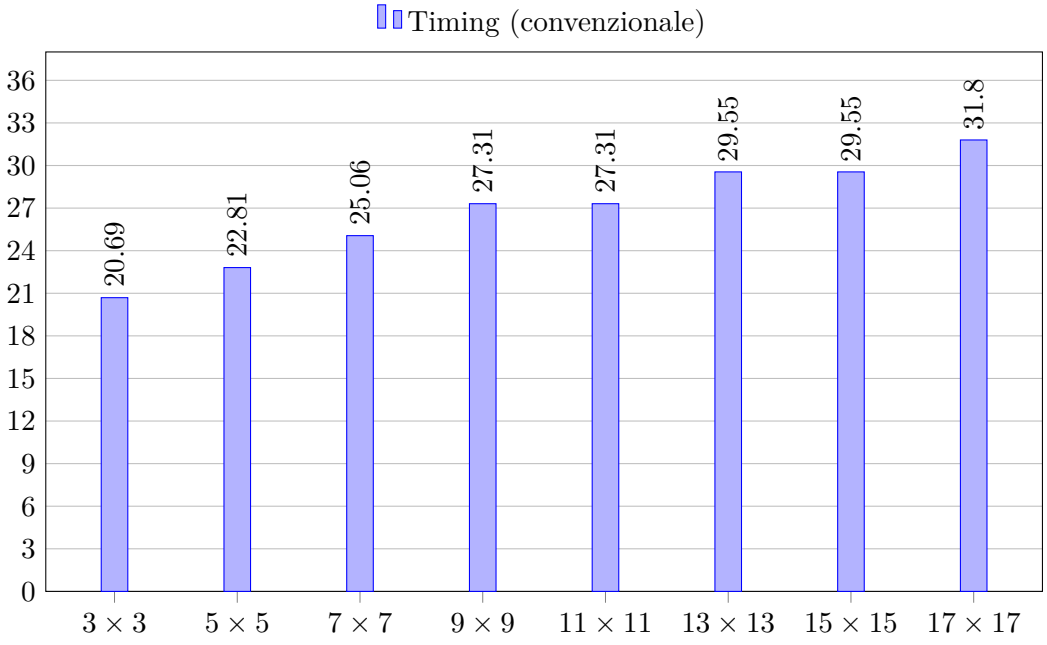

Figura 5.1: Grafico del *timing* per l'algoritmo di convoluzione convenzionale.

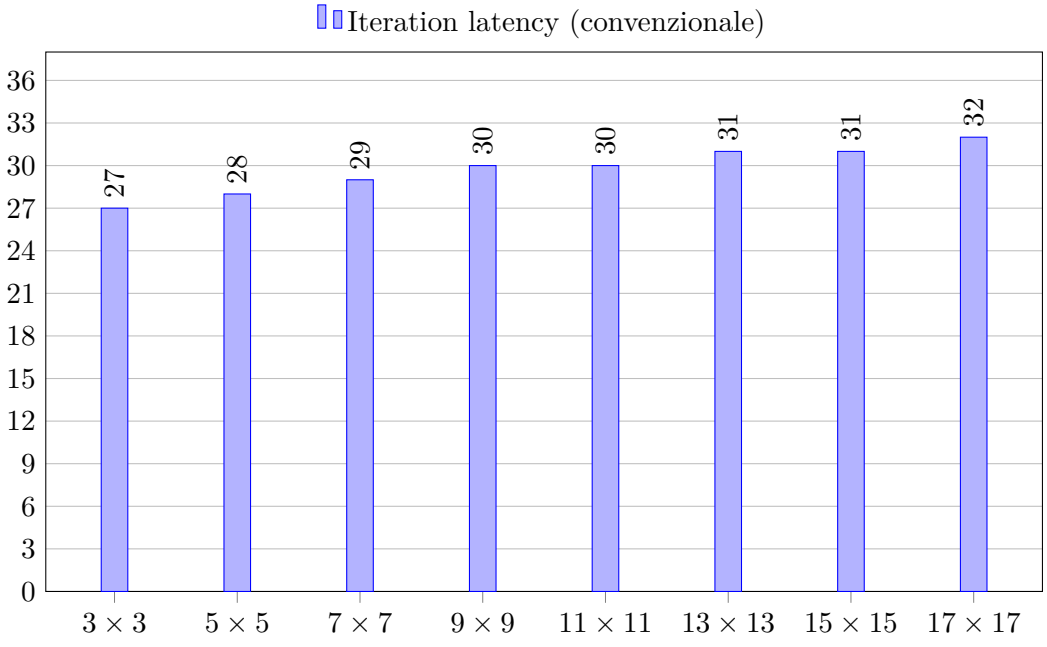

Figura 5.2: Grafico dell'*iteration latency* per l'algoritmo di convoluzione convenzionale.

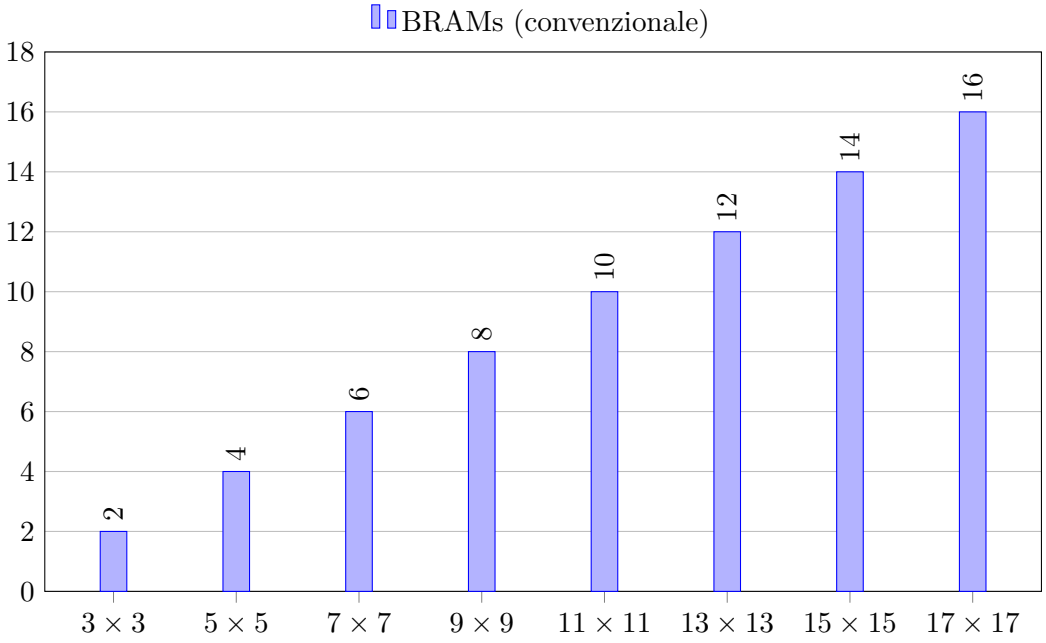

Figura 5.3: Grafico del numero di BRAMs utilizzate dall'algoritmo di convoluzione convenzionale.

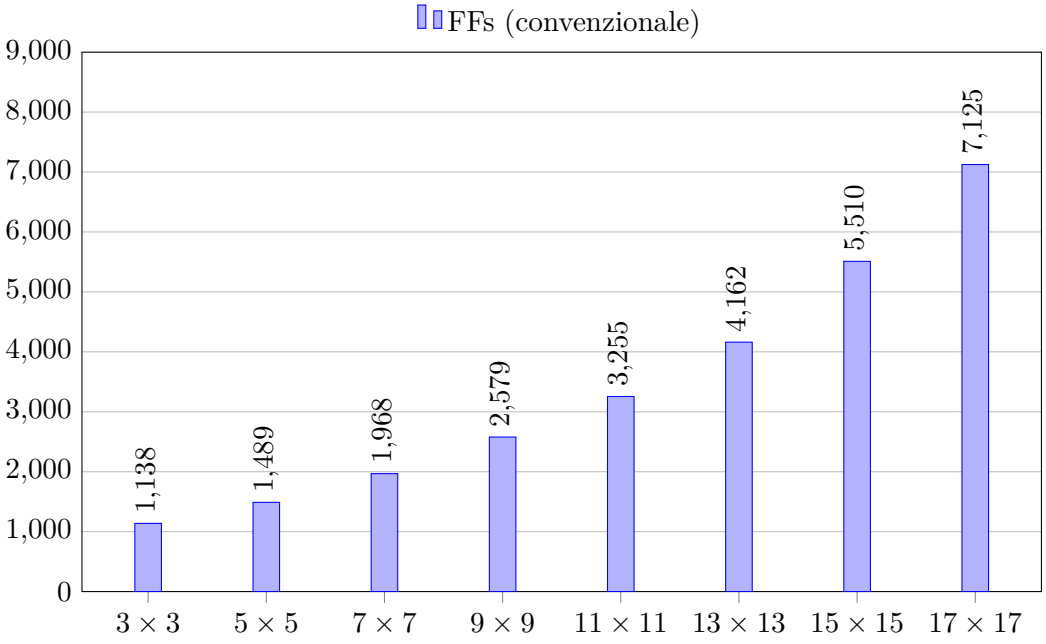

Figura 5.5: Grafico del numero di FFs utilizzati dall'algoritmo di convoluzione convenzionale.

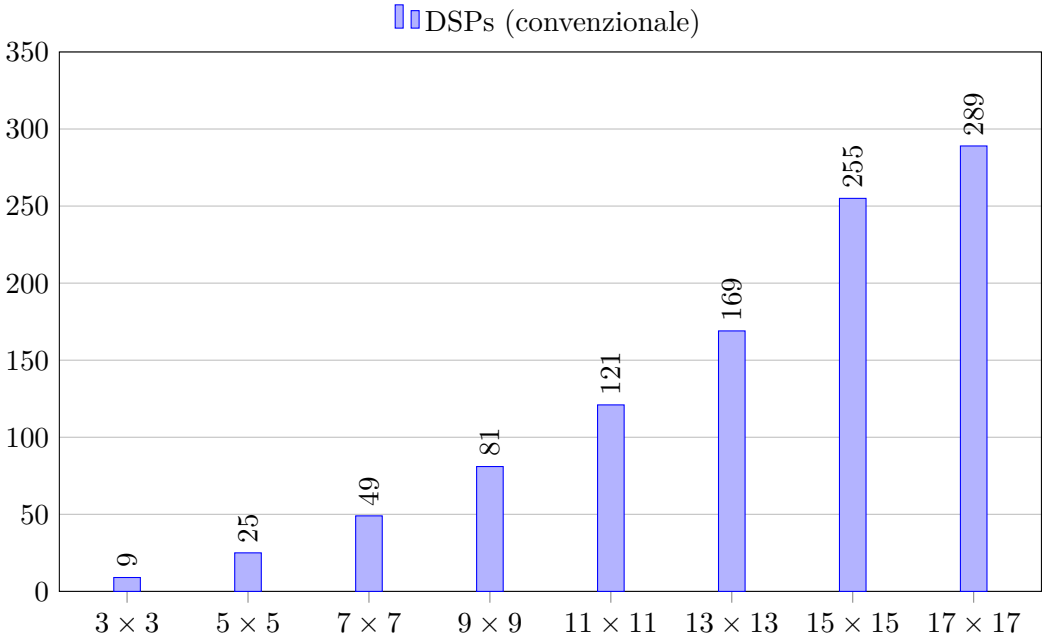

Figura 5.4: Grafico del numero di DSPs utilizzati dall'algoritmo di convoluzione convenzionale.

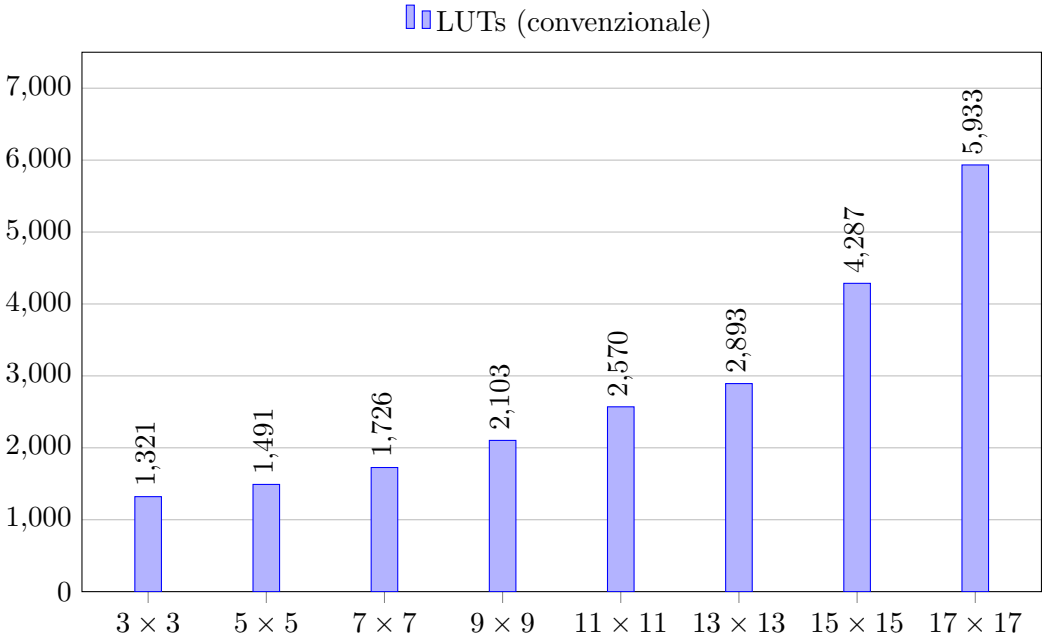

Figura 5.6: Grafico del numero di LUTs utilizzate dall'algoritmo di convoluzione convenzionale.

Come si vedrà nella sezione successiva, l'utilizzo di BRAMs non varierà tra l'implementazione convenzionale e quella separabile, poiché la loro allocazione dipende dalla dimensione dei kernel utilizzati (che risulta analoga per entrambe le implementazioni). Tale aspetto non costituirà pertanto un parametro di ottimizzazione.

5.2 Il nuovo filtro: un confronto

Le prestazioni e le risorse utilizzate dall'algoritmo di convoluzione separabile, nell'implementazione presentata, sono mostrate in Figura 5.7, Figura 5.8, Figura 5.9, Figura 5.10, Figura 5.11 e Figura 5.12, a confronto con le corrispondenti dell'algoritmo convenzionale.

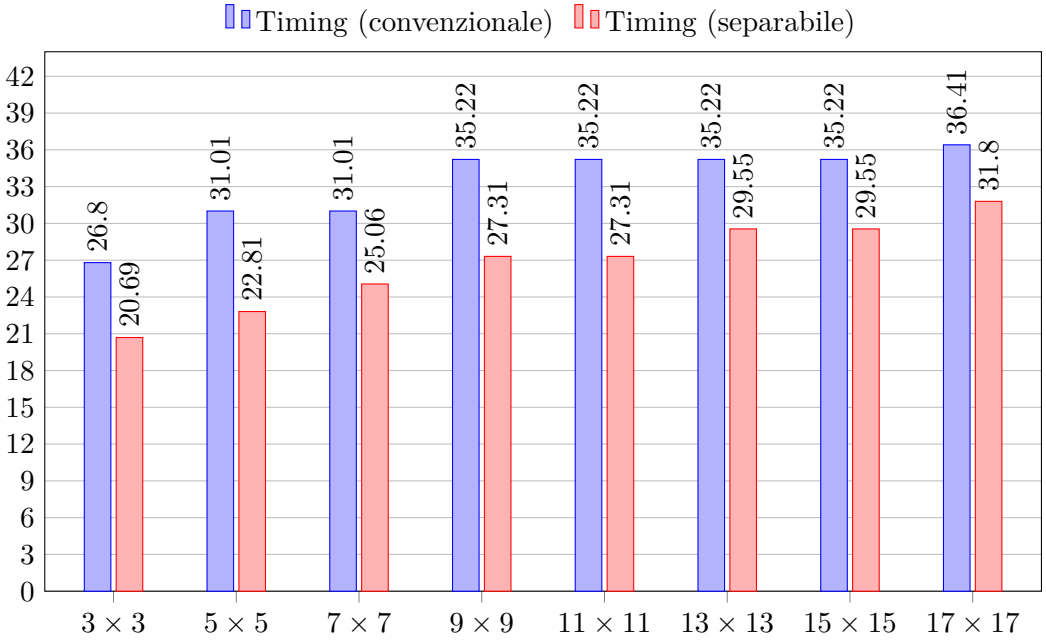

Figura 5.7: Grafico del *timing* per l'algoritmo di convoluzione separabile, a confronto con quello convenzionale.

Come si può evincere dal grafico, il *timing* non costituisce un parametro di ottimizzazione.

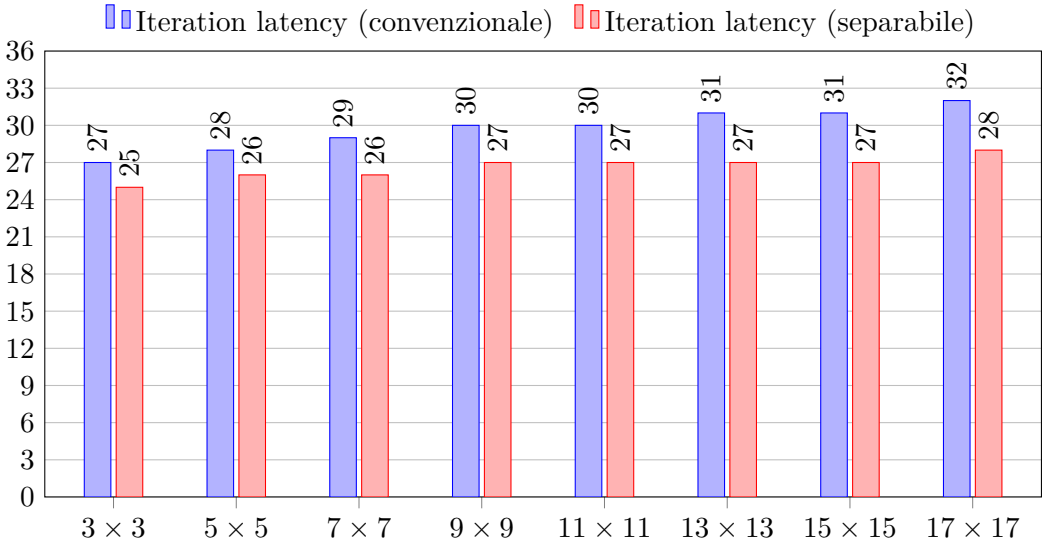

Figura 5.8: Grafico dell'*iteration latency* per l'algoritmo di convoluzione separabile, a confronto con quello convenzionale.

Come si può evincere dal grafico, l'*iteration latency* risulta ottimizzata di 2/4 cicli di clock per ogni dimensione del kernel utilizzato.

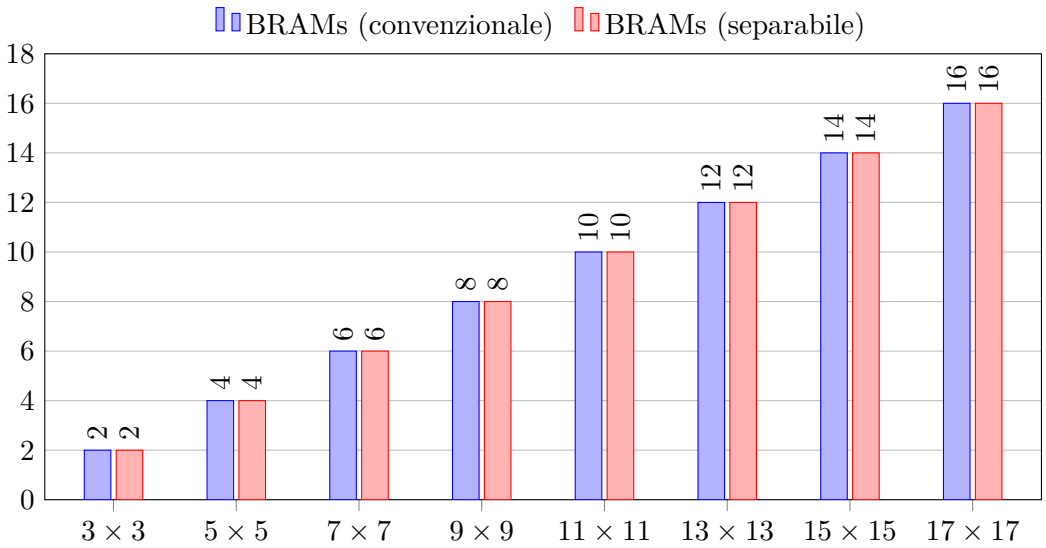

Figura 5.9: Grafico del numero di BRAMs utilizzate dall'algoritmo di convoluzione separabile, a confronto con quello convenzionale.

Come precedentemente accennato, l'utilizzo di BRAMs non costituisce un parametro di ottimizzazione.

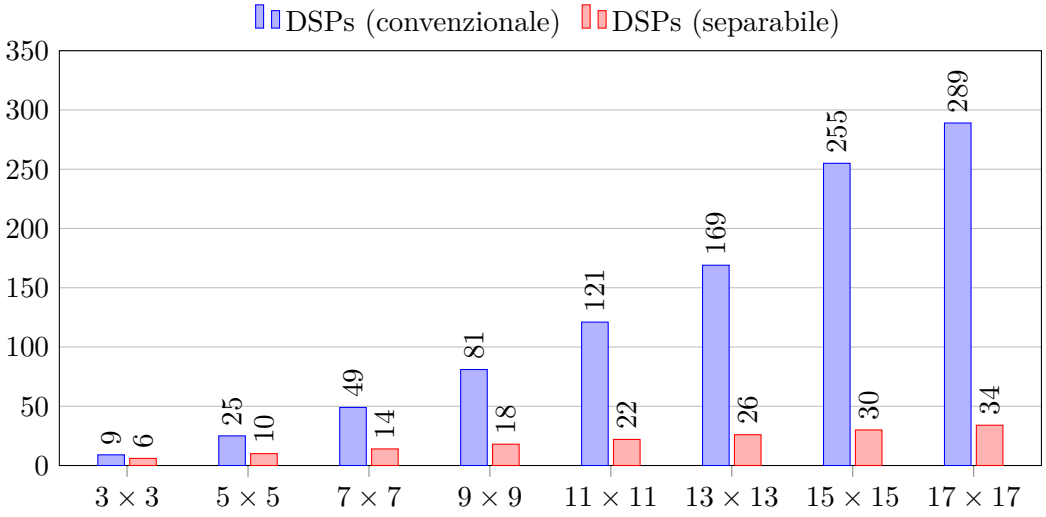

Figura 5.10: Grafico del numero di DSPs utilizzati dall'algoritmo di convoluzione separabile, a confronto con quello convenzionale.

Come si può evincere dal grafico, l'utilizzo di DSPs è drasticamente ridotto, da un minimo del 33%, nel caso 3×3, ad un massimo dell'88%, nel caso 17×17. Esso rappresenta uno dei parametri di maggiore ottimizzazione fornito dall'implementazione separabile.

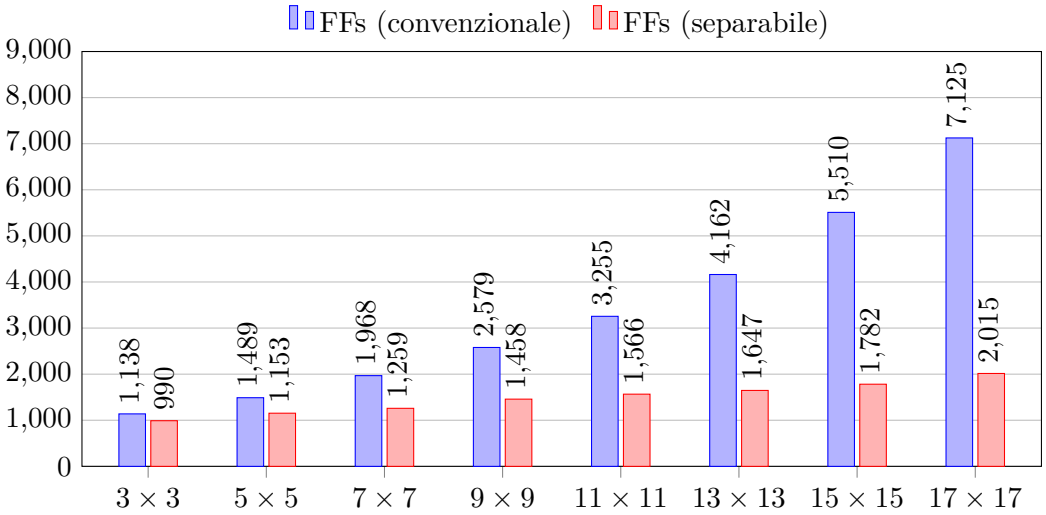

Figura 5.11: Grafico del numero di FFs utilizzati dall'algoritmo di convoluzione separabile, a confronto con quello convenzionale.

Come si può evincere dal grafico, l'utilizzo di FFs è drasticamente ridotto, da un minimo del 13%, nel caso 3×3, ad un massimo del 72%, nel caso 17×17.

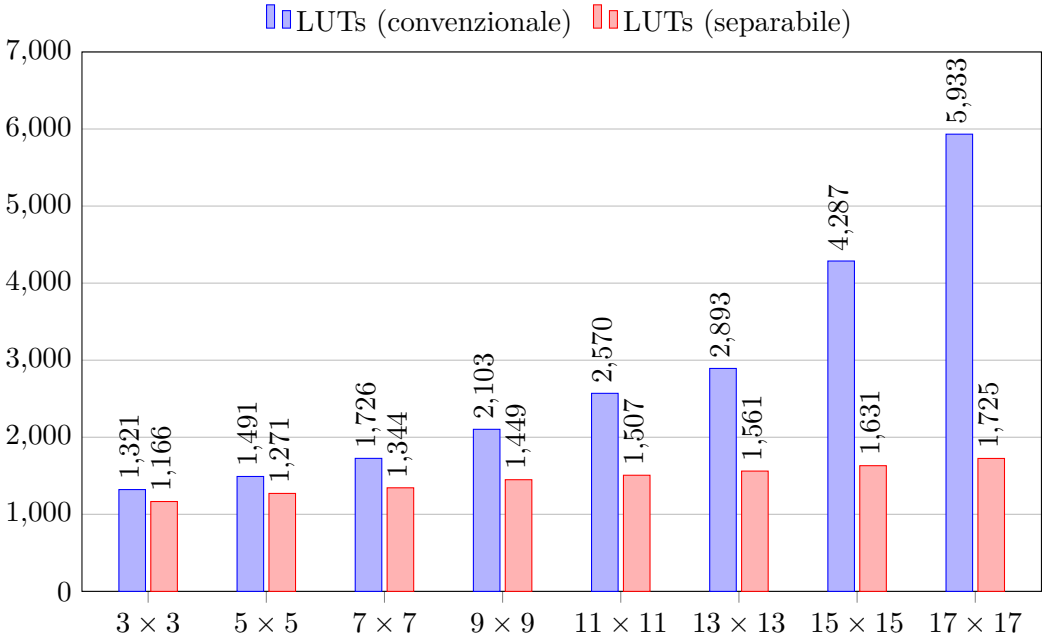

Figura 5.12: Grafico del numero di LUTs utilizzate dall'algoritmo di convoluzione separabile, a confronto con quello convenzionale.

Come si può evincere dal grafico, l'utilizzo di LUTs è drasticamente ridotto, da un minimo del 12%, nel caso 3×3, ad un massimo del 71%, nel caso 17×17.

Si sintetizzano in Tabella 5.1 le differenze percentuali dei parametri mostrati per ogni dimensione del kernel nell'intervallo di riferimento.

	Timing	**Iteration latency**	**BRAMs**	**DSPs**	**FFs**	**LUTs**
3×3	+30%	-7%	0%	-33%	-13%	-12%
5×5	+36%	-7%	0%	-60%	-23%	-15%
7×7	+24%	-10%	0%	-71%	-36%	-22%
9×9	+29%	-10%	0%	-78%	-43%	-31%
11×11	+29%	-10%	0%	-82%	-52%	-41%
13×13	+19%	-13%	0%	-85%	-60%	-46%
15×15	+19%	-13%	0%	-88%	-68%	-62%
17×17	+14%	-13%	0%	-88%	-72%	-71%

Tabella 5.1: Differenze percentuali di prestazioni e risorse utilizzate per ogni dimensione del kernel nell'intervallo di riferimento.

5.2.1 Realizzabilità

Le consistenti ottimizzazioni apportate al filtro di convoluzione dall'algoritmo separabile permettono inoltre l'installazione di un maggior numero di filtri contemporaneamente sul dispositivo. Considerando infatti come risorse disponibili la differenza tra le risorse totali messe a disposizione dal dispositivo e quelle impiegate dal resto della logica necessaria al suo funzionamento, il numero di filtri installabili segue l'andamento illustrato in Figura 5.13.[1]

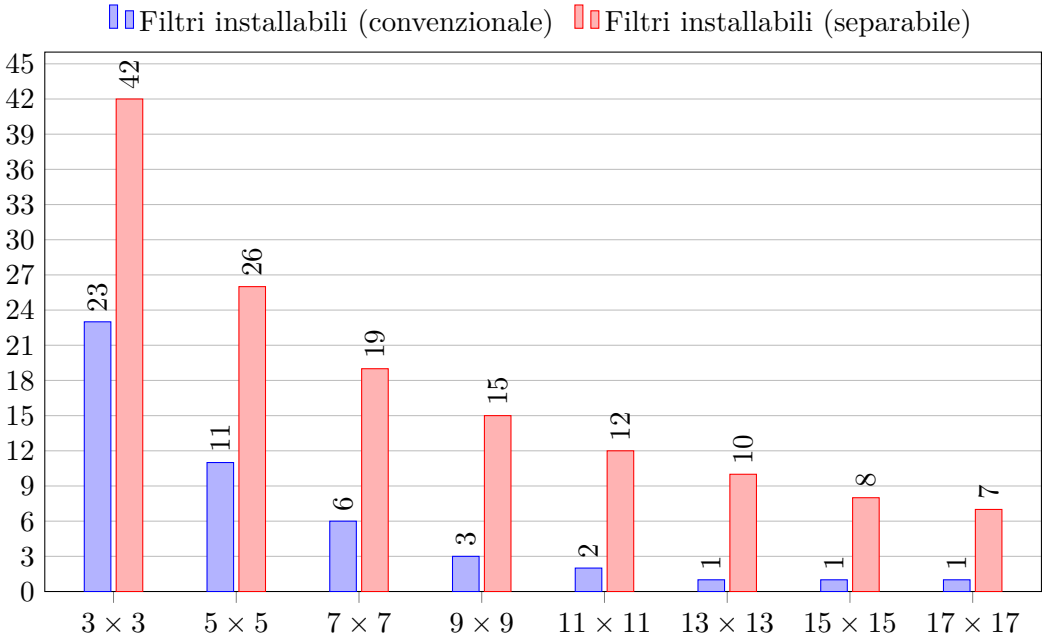

Figura 5.13: Grafico del numero di filtri installabili contemporaneamente sul dispositivo, confrontando l'implementazione separabile con quella convenzionale.

Il numero aggiuntivo di filtri installabili è sintetizzato in Tabella 5.2.

	3×3	5×5	7×7	9×9	11×11	13×13	15×15	17×17
N° di filtri	+19	+15	+13	+12	+10	+9	+7	+6

Tabella 5.2: Numero aggiuntivo di filtri separabili installabili sul dispositivo rispetto alla versione convenzionale.

Poiché tuttavia l'utilizzo pratico del filtro prevede che l'implementazione installata sia in versione configurabile, diversamente da quella precedentemente illustrata, si mostra in Figura 5.14 anche tale confronto.

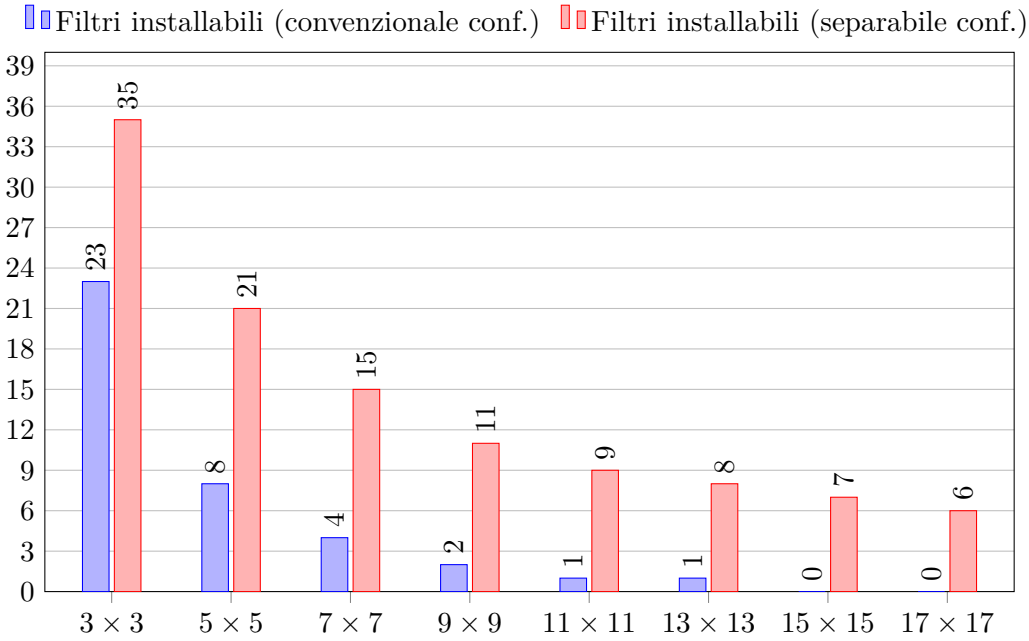

Figura 5.14: Grafico del numero di filtri configurabili installabili contemporaneamente sul dispositivo, confrontando l'implementazione separabile con quella convenzionale.

Il numero aggiuntivo di filtri installabili in versione configurabile è sintetizzato in Tabella 5.3.

	3×3	5×5	7×7	9×9	11×11	13×13	15×15	17×17
N° di filtri	+12	+13	+11	+9	+8	+7	+7	+6

Tabella 5.3: Numero aggiuntivo di filtri separabili configurabili installabili sul dispositivo rispetto alla versione convenzionale.

Capitolo 6

Conclusioni

I risultati sperimentali visti al termine di questo lavoro hanno evidenziato come l'utilizzo di filtri separabili comporti un'ottimizzazione estremamente consistente nell'impiego di risorse di un dispositivo rispetto all'utilizzo di filtri convenzionali. Tale ottimizzazione costituisce pertanto un aspetto di estremo interesse se si desidera accrescere la complessità della logica installata su un dispositivo dotato di risorse limitate come un FPGA. Come accennato nel Capitolo 2 infatti, l'esecuzione, ad esempio, di una logica complessa come quella di una CNN richiede una cospicua quantità di risorse, e ciò costituisce un forte ostacolo se ci si appresta a realizzare un tale obiettivo senza prestare la massima attenzione all'ottimizzazione di ogni singolo aspetto implementativo. L'utilizzo di un algoritmo separabile rappresenta pertanto, oltre che un eccellente strumento di ottimizzazione per qualsiasi applicazione di elaborazione immagini che preveda l'utilizzo di kernel separabili, un elemento di fondamentale importanza per la realizzazione di obiettivi di questa portata.

Tale elemento rappresenterebbe tuttavia uno dei molteplici aspetti che si dovrebbero gestire per il raggiungimento di tale scopo.

Sviluppi futuri di questo lavoro potrebbero compiere questo passo individuando, come presupposto primario, un'architettura adeguata, basata sulla riduzione ai minimi termini della complessità dei livelli interni, al fine di gravare quanto meno risulti possibile sulle limitate risorse disponibili.[1] Compiuto questo passo, si potrà procedere all'implementazione dei moduli necessari al funzionamento efficiente di tale rete; uno tra questi, il filtro separabile.

[1] tra le possibili opzioni attualmente note, *SqueezeNet* può rappresentare una scelta interessante.

Bibliografia

[1] Luca Bonfiglioli. «Progettazione di filtri di convoluzione riconfigurabili per sistema embedded». Università di Bologna, Tesi di laurea. 2017.

[2] Stefano Mattoccia. «Mapping of computer vision algorithms on FPGAs with High Level Synthesis tools». Summer School on Deep Learning on Chip (Macloc2017). Politecnico di Torino, Torino (Italy), set. 2017. URL: http://vision.deis.unibo.it/~smatt/Seminars/Macloc_2017/Convolution_filters_HLS.pdf.

[3] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Gee Hock Ong, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra e Guy Boudoukh. «Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?» In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey, California (USA), feb. 2017, pp. 5–14. DOI: 10.1145/3020078.3021740.

[4] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss e Eric Chung. «Accelerating Deep Convolutional Neural Networks Using Specialized Hardware». Feb. 2015. URL: https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/.

[5] Xilinx. «Vivado Design Suite User Guide». v2017.1. Apr. 2017. URL: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf.

[6] Xilinx. «Zynq-7000 All Programmable SoC Data Sheet: Overview». DS190 (v1.11). Giu. 2017. URL: https://www.xilinx.com/support/documentation/data_sheets/ds190-Zynq-7000-Overview.pdf.

[7] Xilinx. «Zynq-7000 Architecture». URL: https://www.xilinx.com/content/dam/xilinx/imgs/block-diagrams/zynq-mp-core-dual.png.

Ringraziamenti

In questa sezione conclusiva, desidero ringraziare il professor Stefano Mattoccia per avermi dato la possibilità di approfondire l'affascinante mondo della visione artificiale, guidandomi con pazienza ed attenzione lungo tutto il percorso che ha condotto alla realizzazione di questo lavoro.

Ringrazio inoltre i miei colleghi, e amici, Matteo, Marco, Simone e Nicola, per aver reso il percorso di studi condiviso una preziosa ed indimenticabile esperienza di vita.

Ringrazio infine i miei genitori, per avermi dato la possibilità di intraprendere questo percorso, senza i quali non sarebbe mai stato possibile raggiungere questo traguardo.