Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica

SISTEMA DI VISIONE STEREO SU ARCHITETTURA ZYNQ

Relatore:
Prof.
STEFANO MATTOCCIA

Candidato:
RICCCARDO
ALBERTAZZI

I Sessione
Anno Accademico 2015-2016

Contents

Lo scopo della tesi è creare un'architettura in FPGA in grado di ricavare informazioni 3D da una coppia di sensori stereo. La pipeline è stata realizzata utilizzando il System-on-Chip Zynq, che permette una stretta interazione tra la parte hardware realizzata in FPGA e la CPU. Dopo uno studio preliminare degli strumenti hardware e software, è stata realizzata l'architettura base per la scrittura e la lettura di immagini nella memoria DDR dello Zynq. In seguito l'attenzione si è spostata sull'implementazione di algoritmi stereo (rettificazione e stereo matching) su FPGA e nella realizzazione di una pipeline in grado di ricavare accurate mappe di disparità in tempo reale acquisendo le immagini da una camera stereo.

Part I
INTRODUZIONE

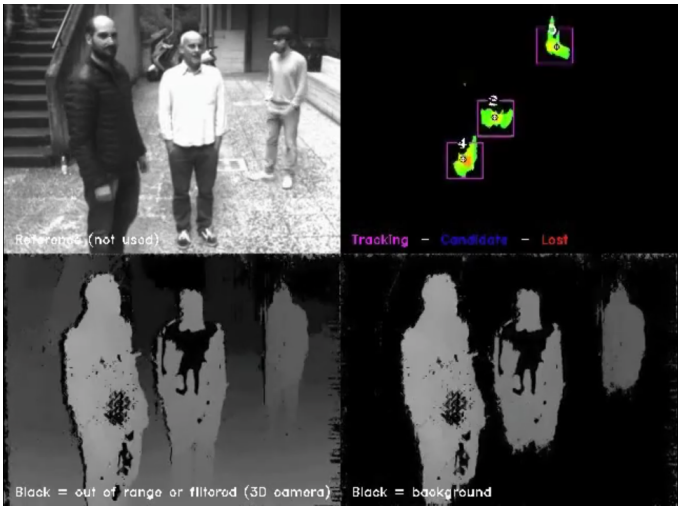
Uno dei grandi obiettivi dell'informatica fin dai suoi albori è stato ricavare informazioni dal mondo esterno in maniera automatica per poter successivamente analizzare tali informazioni e produrre dei risultati significativi per l'uomo. Con l'avvento dei microprocessori e il loro costante miglioramento in termini di prestazoni e consumi sono nate scienze la cui ricerca è intrinsecamente legata all'analisi di una grande mole di dati e che richiedono l'ausilio da parte dell'hardware per poter garantire dei risultati significativi in tempi ragionevoli. In particolare la 3D Vision si occupa di ricavare dati dall'ambiente circostante e sulla natura degli oggetti di cui esso è composto.

Una volta costruiti i sensori di immagine per poter osservare l'ambiente, il primo problema che la visione 3D deve affrontare è il riconoscimento della profondità. In particolare la Stereo Vision cerca di risolvere il problema utilizzando due sensori (che vanno a costituire una *stereo camera*) in modo da ricavare la distanza di un oggetto in base alla posizione che va ad occupare nelle due immagini catturate dai sensori: un oggetto vicino sarà ripreso più a destra dal sensore di sinistra, e più a sinistra del sensore di destra (si faccia una prova banale portando un oggetto vicino al viso e chiudendo un occhio alla volta). Il problema più arduo diventa a questo punto la corrispondenza tra i punti nelle due immagini: dato un punto nell'immagine di sinistra, come è possibile trovare il suo corrispondente nell'immagine di destra?

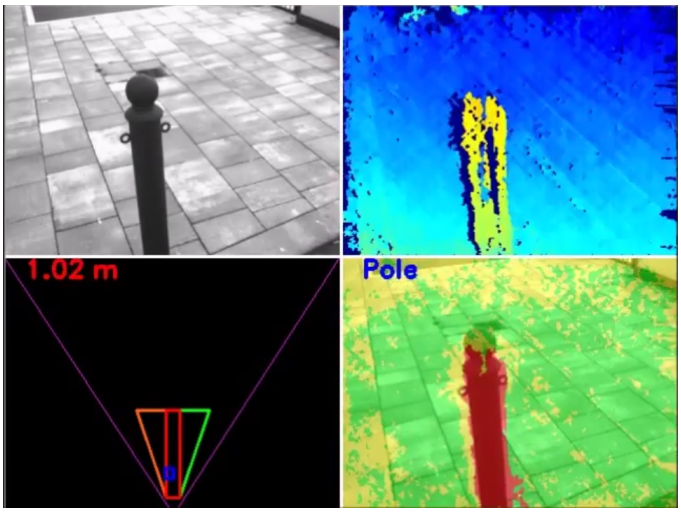
Figure 1: Attraverso la corrispondenza dei punti è possibile ricavare la *mappa di disparità*, un'immagine in cui la profondità, o depth, è codificata attraverso l'intensità del colore; tipicamente a valori più alti di intensità corrisponde una distanza minore.

Esistono numerosi algoritmi che si occupano del cosiddetto *correspondence problem* per ricavare la profondità della scena osservata. Come spesso accade nell'informatica, ad una qualità ed una accuretezza migliori corrispondono delle prestazioni più scadenti; anche gli algoritmi più semplici, eseguiti su un comune microprocessore, non sono in grado di ricavare informazioni in tempo reale. Diventa perciò necessario utilizzare soluzioni hardware più specifiche e personalizzabili per poter implementare questi algoritmi; le GPU sono una prima soluzione e permettono di eseguire questi algoritmi in tempi ragionevoli, ma hanno lo svantaggio di avere dei consumi molto elevati (si può arrivare fino a centinaia di Watt). E' quindi in questo contesto che si colloca la scelta di utilizzare le FPGA, componenti hardware completamente programmabili che consentono di realizzare una soluzione compatta, con prestazioni ottime e con consumi estremamente ridotti (pochi Watt).

In ambito di ricerca il Professore Mattoccia ha già realizzato un sistema di visione stereo real-time basato su FPGA; sulla base di questo sistema sono stati perseguiti altri progetti di ricerca di più alto livello, come sistemi di SLAM, sistemi di tracking, sistemi di guida per veicoli autonomi e sistemi per l'ausilio di persone non vedenti grazie al rilevamento di ostacoli o al riconoscimento di oggetti di uso comune (come banconote o strisce pedonali).



(a)



(b)

Figure 2: Applicazioni della camera stereo per sistemi di tracking (a) e per sistemi di ausilio di persone non vedenti (b)

In particolare la camera realizzata è basata su scheda Spartan di Xilinx, composta da sola FPGA; lo svantaggio di una soluzione basata esclusivamente su FPGA è la necessità di dover sempre utilizzare un dispositivo esterno per poter programmare la camera ed eseguire algoritmi di visione che sono maggiormente portati per una esecuzione su CPU. Inoltre la co-

municazione tra la camera e l'host (attualmente tramite interfaccia USB) rappresenta un ulteriore collo di bottiglia per l'intero sistema e ne degrada le prestazioni.

Lo scopo di questa tesi è di creare un nuovo sistema di visione stereo basato sul System-on-Chip Zynq. Questo chip contiene al suo interno sia una CPU multi-core che una parte di logica programmabile ed è pertanto ideale per ottenere la massima sinergia tra hardware e software, con dimensioni ridotte e consumi comparabili a quelli di un sistema basato su sola FPGA. Avendo a disposizione più risorse hardware rispetto alla scheda già realizzata, sarà inoltre possibile costruire un sistema in grado di rilevare con più accuratezza la profondità (e quindi gli oggetti) e con una velocità di elaborazione maggiore. L'obiettivo è quindi migliorare il riconoscimento 3D sia in termini di qualità che di prestazioni: ciò porterà in maniera diretta dei miglioramenti per i progetti di ricerca di più alto livello.

Part II
STRUMENTI DI SVILUPPO: ZEDBOARD, ZYNQ e VIVADO

Inizialmente sono stati studiati gli strumenti hardware e software per poter lavorare nell'ambiente delle FPGA: dalla ZedBoard e il System-on-Chip Zynq ai protocolli AXI e alla suite Vivado di Xilinx

1 Studio delle tecnologie hardware e software utilizzate

Le sezioni seguenti non pretendono di analizzare in maniera esaustiva ogni aspetto degli strumenti hardware e software utilizzati, ma vogliono solamente fare da preambolo per capire al meglio il lavoro svolto più avanti.

Il nucleo centrale della ZedBoard è il System-on-Chip ZYNQ, la cui architettura generale è formata da due componenti: il Processing System (PS) e la Programmable Logic (PL). Queste parti possono essere usate indipendentemente o insieme ma è la loro combinazione che costituisce il punto di forza maggiore di questo System on Chip.

Il Processing System è costituito da un processore ARM dual-core Cortex-A9, operante alla frequenza massima di 667 MHz e in grado di lavorare sia in modalità multiprocessore che singola.
La Programmable Logic è costituita in maniera predominante da general purpose FPGA, che è formata da vari componenti elementari:

- Lookup Table (LUT): una risorsa flessibile capace di realizzare (i) una funzione logica con sei o meno ingressi; (ii) una piccola Read Only Memory; (iii) una piccola RAM; (iv) un registro a scorrimento. Più LUT possono essere combinate insieme per formare funzioni logiche più complesse.

- Flip Flop (FF): un elemento circuitale sequenziale che implementa un registro di dimensione 1 bit con funzionalità di reset.

- Input/Output Blocks (IOBs): elementi che garantiscono l'interfacciamento tra risorse presenti nella PL e dispositivi fisici esterni. Ogni IOB può gestire un segnale di ingresso o di uscita di 1 bit. Ogni IOB include anche una risorsa IOSERDES per la conversione programmabile tra i formati seriale e parallelo (serializzazione e deserializzazione).

Oltre ai componenti generali sono presenti altri componenti speciali:

- Block RAM (BRAM): possono implementare delle RAM, delle ROM o delle FIFO. Ogni Block RAM può memorizzare fino a 36 Kb di informazioni e può essere configurata sia come singolo dispositivo da 36 Kb che come due dispositivi indipendenti da 18 Kb. Una Block RAM può essere configurata inoltre con una o due porte di accesso.

- DSP48E1s: elementi specifici per l'implementazione di operazioni aritmetiche con operandi di media ed elevata lunghezza.

La PL riceve quattro clock di input dal PS e ha inoltre la capacità di generare e distribuire i suoi segnali di clock in maniera indipendente dal PS.

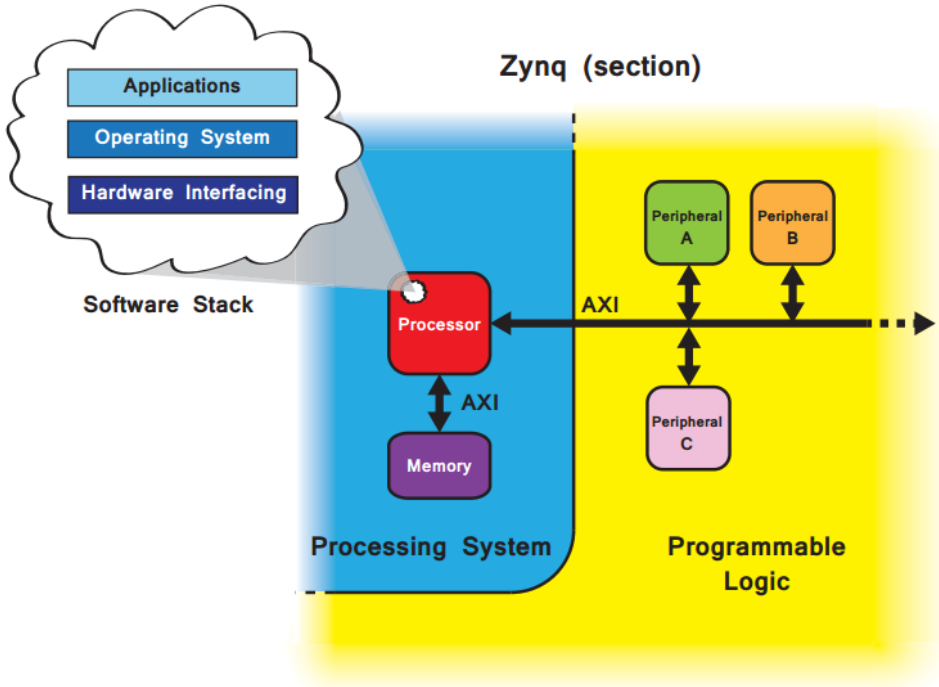


Figure 3: Struttura dello Zynq

La ZedBoard è la evaluating board che racchiude al suo interno il SoC Zynq e contiene una serie di interfacce standard per poter realizzare il design da progettare. Le interfacce di maggiore interesse sono:

- 8 switch
- 8 led
- PMOD per connettere I/O generico
- Uscita VGA a 4 bit
- Porta Ethernet
- SDCard reader

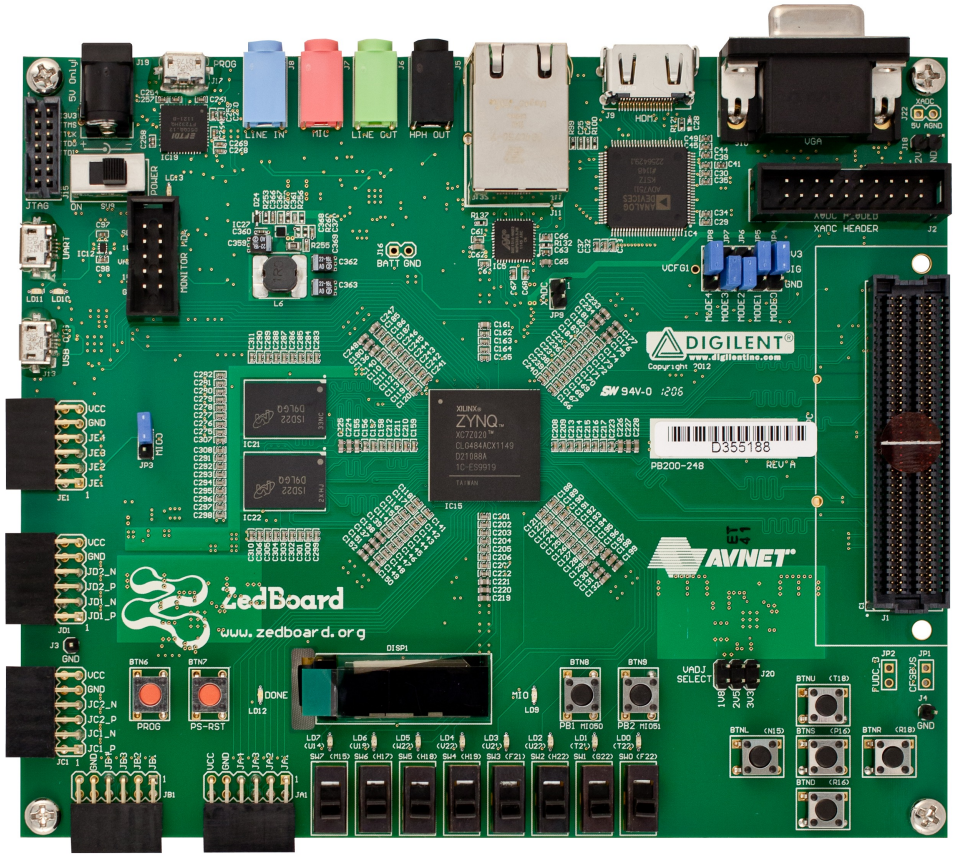


Figure 4: La ZedBoard

1.1 Protocolli AXI e comunicazione PL-PS

AXI o Advanced eXtensible Interface è uno standard di comunicazione su bus definito da ARM e utilizzato da Xilinx all'interno del SoC Zynq. In particolare si hanno due tipologie di protocolli AXI:

- **AXI Memory-Mapped** E' il protocollo più completo in quanto permette la trasmissione dei dati tramite indirizzamento. Il protocollo prevede che più interfacce master (che generano gli indirizzi e sono reponsabili della comunicazione) possano comunicare con più interfacce slave (il cui compito è soltanto rispondere all'interazione richiesta dal master). L'interfaccia è composta da 5 canali di comunicazione: write-

address-channel (canale dal master allo slave per trasmettere l'indirizzo di scrittura), write-data-channel (canale dal master allo slave per trasmettere il dato da scrivere), write-response-channel (canale dallo slave al master per rispondere una volta effettuata la scrittura), read-address-channel (canale dal master allo slave per trasmettere l'indirizzo di lettura), read-data-channel (canale dallo slave al master per trasmettere il dato richiesto). Il protocollo è presente in una versione full e in una versione lite, indicate, rispettivamente, per trasferimenti rapidi di grandi quantità di dati e piccoli trasferimenti di dati a bassa velocità (ad esempio per parametri di configurazione di moduli hardware).

- **AXI Stream** A differenza dell'AXI MM, il protocollo AXI Stream prevede la trasmissione di dati in maniera seriale da un endpoint master ad un endpoint slave. Il protocollo non prevede l'uso di indirizzi ed è perciò molto più efficiente ed indicato per la costruzione di filtri o di pipeline di elaborazione dati in cui si prevede che l'uscita di un modulo diventi l'ingresso di un altro modulo. L'interfaccia AXI Stream minimale (utilizzata nei moduli realizzati) è formata da un unico canale contenente i dati e due soli segnali di handshake: un segnale di READY che va dallo slave al master, che indica che lo slave è pronto a ricevere il dato, e un segnale di VALID che va dal master allo slave, che indica che il master è pronto a inviare il dato; la trasmissione del dato è ritenuta valida se nello stesso clock sia VALID che READY sono abilitati.

Per permettere la comunicazione da PL a PS sono presenti 4 porte HP (High Performance) con interfacce AXI Full Slave a 64 bit, mentre per la comunicazione da PS a PL è presente una porta GP (General Purpose) con interfaccia AXI Full Master. Tramite le porte HP è possibile scrivere nella memoria DDR, che appartiene alla parte PS del SoC. Tramite la porta GP è possibile per esempio customizzare i moduli sintetizzati su FPGA.

1.2 OV7670

L'OV7670 è un sensore di immagine low-cost prodotto dalla OmniVision e facilmente utilizzabile: infatti per poter essere utilizzato nella configurazione di default non ha bisogno di alcun tipo di programmazione.

Figure 5: OV7670

I segnali di input/output del sensore sono i seguenti:

- 3V3: tensione a 3.3V
- GND: massa
- XCLK (input): clock di ingresso del sensore; i valori ammissibili sono tra 10 MHz e 48 MHz, con valore tipico (qui utilizzato) di 24 MHz.
- RESET (input): reset del sensore, attivo basso.
- PWDN (input): power down, di default a 0.
- PCLK (output): il pixel clock con cui il sensore invia i dati in uscita. Di default è uguale a XCLK.
- VSYNC (output): vertical synchronization, viene settato a 1 prima dell'inizio di un frame e dopo la fine del frame.
- HREF (output): horizontal synchronization, rimane attivo per tutto il tempo in cui viene trasferita una riga di un'immagine.
- D[7..0] (output): dati generati dal sensore (si veda il significato più avanti).

Sono presenti altri due segnali utilizzati per la programmazione del sensore (qui non utilizzati) e che usano il protocollo di comunicazione seriale SCCB (equivalente a I2C):

- SDIOC (input): clock usato nella comunicazione (valore tipico 400 kHz).
- SDIOC (input-output): linea seriale per lo scambio dei dati.

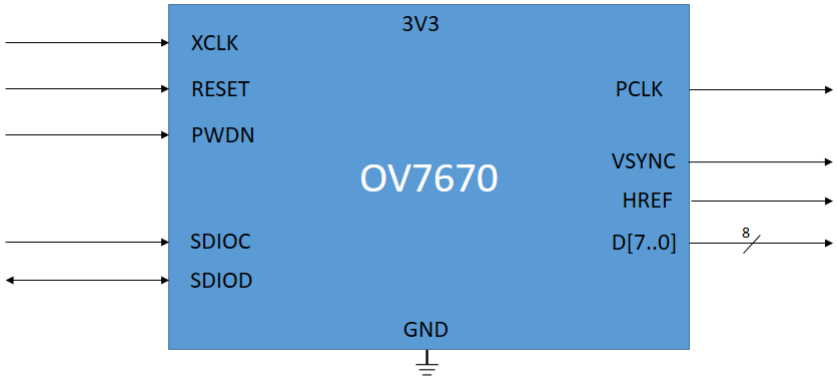


Figure 6: Interfaccia di I/O dell'OV7670

Di default il frame generato dal sensore è a risoluzione VGA (640x480 pixel) e il formato di uscita é YCbCr; in questo formato un pixel viene codificato utilizzando tre componenti, uno per la luminanza (Y, corrisponde all'intensità in grayscale) e due componenti per la crominanza (Cb e Cr). In particolare il sensore OV7670 utilizza un byte per ogni componente e condivide le componenti di crominanza tra due pixel consecutivi, consentendo di memorizzare un pixel utilizzando in media 2 byte. L'ordine temporale di uscita delle tre componenti è il seguente:

N	Byte
1st	Cb0
2nd	Y0
3rd	Cr0
4th	Y1
5th	Cb2
6th	Y2
7th	Cr2
8th	Y3
...	...

Figure 7: Ordine di uscita dei valori YCbCr nell'OV7670 a default

Come si può notare per utilizzare i sensori in grayscale è sufficiente campionare i dati dispari in uscita del sensore.

1.3 Sensori Stereo Aptina

I sensori prodotti da Aptina acquisicono immagini a risoluzione VGA (fino a un massimo di 752x480 pixel) ad un frame-rate massimo di 60 fps. La peculiarità di questi sensori è che possono essere usati in maniera stereo, ovvero atraverso un'opportuna programmazione è possibile fare in modo che ad ogni clock vengano emessi in output pixel corrispondenti. I sensori sono programmabili attraverso protocollo I2C (attraverso opportune librerie già realizzate) e utilizzano il protocollo seriale LVDS per trasmettere in uscita il clock e i dati. In particolare per ogni coppia di pixel (ogni pixel è a 8 bit) vengono emessi 18 bit (bit start - 16 bit per i pixel - bit end) ad una frequenza di clock 18 volte superiore. Il vantaggio di questa configurazione è che vengono utilizzati pochi pin (4 contro almeno 18). Lo svantaggio è che la ZedBoard non dispone direttamente di pin clock-capable ed è stato quindi necessario predisporre un setup hardware ad hoc per poter utilizzare i sensori.

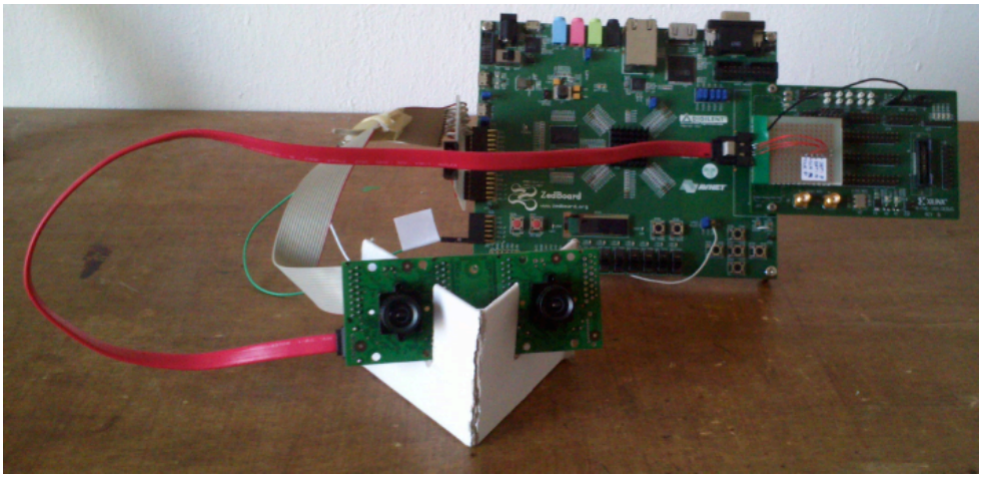


Figure 8: Sensori stereo Aptina e setup della ZedBoard

1.4 Vivado HLS

Vivado HLS è il software di Xilinx che permette di realizzare degli IP Core nei linguaggi di alto livello C/C++. Il software permette infatti di tradurre codice C/C++ in codice HDL (VHDL, Verilog) per costruire dei moduli hardware. Un modulo in HLS è definito attraverso la cosiddetta top-function, cioè la funzione che fa da interfaccia verso il mondo esterno.
La progettazione e il testing del modulo realizzato in Vivado HLS si compone di tre fasi: simulazione, sintesi e cosimulazione. Attraverso la simulazione è possibile testare immediatamente il proprio modulo compilando il proprio codice ed eseguendolo in un normale ambiente C/C++ Eclipse-based. Nella fase di sintesi il codice viene tradotto in un modulo HDL ed è possibile analizzare i tempi di latenza del modulo e le risorse da esso occupate. Nella fase di cosimulazione viene testato il modulo HDL con gli stessi input usati nella simulazione e si verifica che i risultati ottenuti siano gli stessi.

1.4.1 Il linguaggio C++ in Vivado HLS

E' importante notare come ci siano alcune limitazioni nell'uso del linguaggio C++ per creare la top-function desiderata. Innanzitutto al momento della sintesi il tool deve conoscere quante risorse allocare all'interno della logica programmabile, quindi é assolutamente vietato l'uso di memoria dinamica (malloc, free) e l'uso di system call o funzioni standard del linguaggio (come

la printf) che usano all'interno memoria dinamica. Si noti che questa limitazione non è da intendere come un difetto, ma bisogna ricordarsi che stiamo programmando hardware e quindi l'uso di system call e memoria dinamica è totalmente inappropriato e privo di senso in questo contesto. Un'ulteriore limitazione è quella di non usare funzioni ricorsive, in quanto non è possibile sapere a priori quante volte tale funzione sarà chiamata e quindi quante risorse allocare.
Le limitazioni appena esposte valgono solamente per la top-function e tutte le funzioni ad essa afferenti: nel codice sorgente usato per la simulazione (il testbench) è possibile usare tutte le funzioni standard del C e la memoria dinamica.

1.4.2 La sintesi: come viene realizzata la top-function

Il modulo realizzato dal processo di sintesi è una rete sincrona dotata di clock e reset sincrono. Di default la rete è composta secondo il pattern datapath + control unit; nella parte di datapath sono presenti tutti i segnali di input e output definiti dall'utente nella top-function e la parte di elaborazione, mentre nella control unit sono presenti alcuni segnali di controllo inseriti automaticamente durante la sintesi e che servono per controllare la rete realizzata. I segnali della control unit sono:

ap_start (input) nel momento in cui viene settato a 1 il modulo comincia la computazione

ap_idle (output) settato a 1 quando non ha ancora cominciato l'elaborazione

ap_done (output) settato a 1 ogni volta in cui il modulo termina l'elaborazione (in linguaggio di alto livello ogni volta in cui la top-function raggiunge una return)

ap_ready (output) settato a 1 quando il modulo è pronto per elaborare il prossimo input

Di default i tipi di dato primitivi (interi, char) vengono associati ad un segnale booleano chiamato valid che verrà settato a 1 durante i cicli di clock in cui il segnale è valido.

1.4.3 Direttive HLS

Tramite opportuni comandi scritti nel codice, detti direttive, é possibile imporre un determinato comportamento del modulo o di una sua parte oppure indicare una determinata interfaccia per i segnali di ingresso e di uscita. Sono qui riportate alcune delle direttive più importanti:

- **#pragma HLS INTERFACE ap_ctrl_none port=return**
  Usata per non includere la parte di control unit nella rete realizzata; tipicamente la control unit risulta superflua, a meno di necessità di debug (per esempio per effettuare la cosimulazione è necessario che questa direttiva non venga utilizzata).

- **#pragma HLS INTERFACE ap_none port=nome_parametro**

  Usata per chiedere di non inserire ulteriori segnali di controllo (come il segnale di valid) ad un parametro di input/output della top-function.

- **#pragma HLS INTERFACE axis port=nome_parametro**
  Usata per per richiedere che un determinato parametro di input/output della top-function venga sintetizzato secondo il protocollo AXI Stream.

- **#pragma HLS INTERFACE m_axi port=base_addr** Usata per sintetizzare una determinata variabile puntatore secondo il protocollo AXI Full master.

- **#pragma HLS PIPELINE II=NUM_CYCLES**
  Con la direttiva PIPELINE è possibile imporre l'esecuzione di più operazioni legate tra loro in pipeline piuttosto che in maniera sequenziale, in modo da ridurre la latenza totale. Attraverso il parametro II è possibile impostare il throughput desiderato. Se la direttiva è inserita all'interno della top-function si specifica che venga prodotto un risultato ogni NUM_CYCLES cicli di clock, mentre se inserita all'interno di un loop si specifica che il codice interno al loop produca un risultato ogni NUM_CYCLES cicli di clock. Il parametro II è opzionale e viene settato di solito a 1.

- **#pragma HLS RESET variable=nome_variabile**
  Da usare su una variabile statica definita internamente alla top-function;

si ricorda che una variabile statica in C nel momento in cui viene invocata una funzione assume il valore che aveva all'invocazione precedente della funzione stessa, operando quindi da elemento di memoria. Se la variabile statica è stata inizializzata nel momento in cui è stata dichiarata, questa direttiva garantisce la corretta inizializzazione della variabile nel momento in cui viene dato il reset al modulo.

- **#pragma HLS RESOURCE variable=buffer core=RAM_2P_BRAM**
  Le direttive RESOURCE permettono di mappare una variabile su una determinata risorsa. In questo caso la variabile buffer viene mappata in una Block RAM bi-porta.

- **#pragma HLS ARRAY_PARTITION variable=buffer complete dim=1**
  La direttiva permette di partizionare un array su più risorse hardware in modo da poter accedere in parallelo a più elementi dell'array stesso. Utilizzando il parametro complete dim=1 si richiede che l'array sia partizionato completamente sulla sua prima dimensione (in modo da poter accedere in un clock a tutti gli elementi dell'array).

1.4.4 Uso del protocollo AXI Stream in Vivado HLS

Come detto sopra, per creare un'interfaccia AXI Stream a partire da un segnale di input o di output é necessario includere la specifica direttiva axis. Di default viene realizzata una porta AXI Stream con i segnali di controllo minimi per garantire il funzionamento del protocollo, ovvero TVALID (diretto dal master allo slave) e TREADY (diretto dallo slave al master). Riguardo al tipo di dato da usare, è possibile effettuare due scelte:

1. Utilizzare un array, a patto di accedere in maniera sequenziale a tale array (altrimenti si ha errore durante la sintesi). Una firma possibile di una top-function che implementa un filtro di elaborazione è la seguente:

   ```
   void filter(DATA_IN input[IMG_HEIGHT*IMG_WIDTH],
   DATA_OUT output[IMG_HEIGHT*IMG_WIDTH]);
   ```

2. Utilizzare la classe stream: tale classe è messa a disposizione da Vivado HLS (tramite l'header hls_stream.h) e implementa una semplice interfaccia sulla quale invocare metodi quali read (per stream in ingresso) e write (per stream in uscita). Sebbene utilizzare questa classe

implichi una portabilità del codice inferiore, essa risulta più semplice nell'utilizzo: si è obbligati ad accedere in maniera sequenziale allo stream e se si vuole utilizzare più volte lo stesso dato è necessario bufferizzarlo all'interno del modulo. In questo caso la firma per un filtro AXI Stream diventa:

```
void filter(hls::stream<DATA_IN> &inputStream,
hls::stream<DATA_OUT> &outputStream);
```

Nel caso fosse necessario usare il protocollo AXI Stream completo (con segali TLAST, TUSER, ecc..) è possibile usare come tipo di dato dell'interfaccia due strutture dati definite da Vivado HLS nell'header ap_axi_sdata.h (ap_axis per dati signed e ap_axiu per dati unsigned) e personalizzabili per quanto riguarda la dimensione dei dati e degli altri segnali di controllo.

1.4.5 AXI Stream non bloccante

Il protocollo AXI Stream, così come la sua implementazione in Vivado HLS, è bloccante; ciò significa che nel momento in cui viene richiesta una lettura (attraverso una read() per esempio), questa bloccherà il flusso di esecuzione finché non sarà disponibile un dato in ingresso. Viceversa, per uno stream di output, la write() bloccherà il modulo finché il dato in uscita non sarà letto dal modulo a valle (asserendo il segnale TREADY). In alcuni casi è necessario modificare questo comportamento, implementando letture o scritture non bloccanti, cioè operazioni che in ogni caso non bloccano il flusso di esecuzione ma, in caso di lettura o scrittura non effettuata, ritornano un valore di insuccesso.

In Vivado HLS ciò è realizzabile attraverso due metodi della classe hls::stream, read_nb e write_nb (dove nb sta per non-blocking). Nella letteratura sui protocolli AXI di Xilinx viene però affermato che il segnale di READY può dipendere dal segnale di VALID, ma non viceversa. Ciò implica che soltanto le letture non bloccanti sono supportate, mentre le scritture non bloccanti no. Probabilmente la scelta nasce dalla volontà di evitare situazioni di deadlock: se il modulo master effettua una scrittura non bloccante e il modulo slave una lettura non bloccante, il trasferimento non avverrà mai in quanto nessuno dei due moduli asserirà il prorprio segnale di controllo.

1.4.6 Uso del protocollo AXI Full Master in Vivado HLS

Oltre ad utilizzare la specifica direttiva, per utilizzare il protocollo AXI Full Master è necessario definire una variabile puntatore sulla quale effettuare le operazioni di scrittura e di lettura. Il tipo di dato definisce la width dei dati trasferiti (32 o 64 bit). Per effettuare trasferimenti burst è necessario usare la funzione standard del C memcpy, utilizzando il puntatore definito sopra come destinazione per operazioni di scrittura o come sorgente per operazioni di lettura. Un modulo che si interfaccia con il mondo esterno solo attraverso un AXI Master a 64 bit avrà una firma di questo tipo:

```
void my_axim(volatile unsigned long long *base_addr);
```

Per tipi di dato inferiori a 32 bit (ad esempio dati a 8 o 16 bit), al momento del trasferimento i dati vengono impacchettati in un unico vettore di 32 bit e poi spediti con un unico trasferimento. In questo modo con dati a 8 bit è possibile con un solo trasferimento burst trasferire 4*256 = 1024 byte. Per massimizzare le performance in caso di trasferimenti ingenti è consigliato bufferizzare i dati di dimensione inferiore in un vettore della dimensione del trasferimento che si vuole effettuare (a 32 o 64 bit) in modo da evitare la fase di impacchettamento dei dati da parte del protocollo.

1.5 Vivado

Vivado è il cuore dell'ecosistema Xilinx per quanto riguarda la programmazione delle FPGA. Il softwware permette infatti di creare dei progetti a partire da IP in linguaggio HDL o generati tramite Vivado HLS per realizzarne una sintesi in FPGA attraverso le fasi canoniche di sintesi, implementazione e generazione del bitstream con il quale programmare la FPGA.

Per inserire nel progetto un IP generato con Vivado HLS è sufficiente indicare il path del progetto HLS all'interno di Vivado. Una volta settato, l'IP verrà trattato come un normale modulo in HDL.
Oltre ai moduli custom è possibile utilizzare una vasta gamma di IP proprietari di Xilinx: tutti questi IP supportano i protocolli AXI e ne facilitano l'utilizzo. Tra i più importanti si ricordano:

- **Zynq Processing System**: Rappresenta l'astrazione della parte PS del SoC Zynq e rappresenta quindi il ponte di comunicazione tra PL e PS. Il modulo è completamente personalizzabile e permette per esempio di abilitare selettivamente le quattro porte HP, la porta GP, il

protocollo I2C sui pin di MIO desiderati o gli interrupt in ingresso al PS.

- **Processing System Reset**: E' un modulo che permette di generare reset sincroni (attivi alti e bassi) a partire da un reset asincrono e da un clock; la maggior parte dei moduli Xilinx e gli IP generati con HLS (a default) richiedono infatti un reset sincrono. Spesso si utilizza un reset asincrono esterno (per esempio uno switch) e un modulo Processing System Reset per ogni clock presente all'interno del progetto.

- **AXI Memory Interconnect**: E' un modulo inserito di default dal tool quando si connette un modulo AXI Master a un modulo AXI Slave e rappresenta l'astrazione del bus di comunicazione tra i due moduli. Grazie a questo IP è possibile infatti gestire molto velocemente l'instradamento e il mapping di più master ad uno stesso slave, di più slave ad uno stesso master, o di più master su più slave.

- **AXI Stream Data Fifo** Tramite questo modulo è possibile realizzare in maniera efficiente una fifo axi stream impostando la depth. Inoltre risulta molto utile per effettuare il "clock domain crossing", cioè per far interagire moduli che lavorano a frequenze di clock differenti. Per realizzare una fifo asincrona è sufficiente impostarla da GUI e collegare ad essa il clock master (dello stream in uscita) e il clock slave (dello stream in ingresso).

- **AXI GPIO** Questo modulo di General Purpose Input/Output permette di collegare segnali qualsiasi in ingresso o in uscita (come led, switch o altri segnali custom) e di essere acceduti tramite protocollo AxiLite. E' inoltre presente l'opzione di generazione automatica degli interrupt ogni volta che il valore del segnale in ingresso viene modificato.

Una volta realizzato il progetto per via grafica è possibile attraversare le fasi di sintesi ed implementazione per generare infine il bitstream.

1.5.1 ILA Debugger

In Vivado è inoltre possibile effettuare un debug hardware attraverso ILA. Per realizzare ciò verranno inserite dal tool delle risorse aggiuntive dedicate al debug che consentono di memorizzare un numero limitato (impostato

dall'utente) di cicli di clock relativamente ai segnali che si vogliono debuggare. Per effettuare il debug di un segnale basta semplicemente abilitare la voce "Mark as Debug" sul segnale dalla GUI; una volta effettuata la sintesi è necessario eseguire i passi alla voce "Set up Debug" per selezionare i segnali desiderati (tra quelli marcati precedentemente) e la depth dei buffer necessari a memorizzare i valori dei segnali. Una volta caricato il bitstream, dall' "Hardware Manager" sarà possibile visualizzare le forme d'onda dei segnali debuggati. E' possibile far partire il debug in maniera "immediata" (al momento della pressione di un bottone) oppure al verificarsi di una condizione dei segnali a runtime; si parla in quest'ultimo caso di trigger, ed è possibile inserire condizioni quali =, !=, >, < o una serie di condizioni in AND o in OR tra di loro.

1.6 Xilinx SDK

Per sfruttare completamente le potenzialità del SoC ZYNQ è necessario anche poter interagire con la parte PS e programmare i processori ARM. Xilinx SDK è il software eclipse-based il cui scopo è gestire la parte PS e armonizzarne il comportamento con il bitstream della parte PL generato in Vivado.

Il collegamento tra Vivado e SDK è proprio il bitstream: una volta generato, è possibile esportarlo e lanciare un'istanza di SDK che si occuperà di creare un progetto BSP (Board Support Package), il cui scopo è creare uno strato minimo di astrazione per poter interagire in maniera ottimale con l'FPGA.

Una volta generato il BSP è possibile creare un progetto in C (vuoto o a partire da dei template) che verrà eseguito su ARM. Tra le utili funzionalità che il software propone si ricordano:

1. Ridirezione dell'output del programma (printf o xil_printf) via USB in modo da poter essere letto dall'host su cui risiede SDK.

2. Lettura e scrittura di valori nello spazio di indirizzamento Zynq (DDR o FPGA) tramite apposite funzioni mrd e mwr, utilizzate con la console XMD.

3. Dump della memoria, che permette di salvare sull'host una porzione di memoria della scheda (impostando indirzzo di partenza e dimensione in byte).

4. Generazione automatica di driver di supporto per i moduli scritti in Vivado HLS che utilizzano protocolli AxiLite Slave e che vengono pilotati dalla parte PS (tipicamente tramite porta GP). Questi driver permettono di astrarre dall'indirizzo in memoria in cui sono mappati i moduli e dall'indirizzo dello specifico registro; al posto degli indirizzi vengono usate delle costanti e delle funzioni ad hoc get/setValore che internamente effettuano letture o scritture ai registri desiderati.

Part III
TRASFERIMENTO DELLE IMMAGINI IN DDR

Una volta ottenute le conoscenze base sugli strumenti hardware e software è stato realizzato su FPGA il nucleo dell'architettura per il trasferimento nella memoria DDR delle immagini generate dai sensori. Il sistema consente di scrivere e leggere immagini in memoria e invia un interrupt al Processing System ogni volta che un'immagine è stata scritta in DDR.

2 Implementazione dei moduli di trasferimento in DDR

La struttura da realizzare comprende due moduli, uno che si occupa di scrivere un frame e l'altro che si occupa di leggerlo. I frame inoltre non devono essere scritti sempre nella stessa area di memoria, ma devono essere scritti in un numero limitato di frame buffer. E' necessario inoltre che il modulo di lettura possa comunicare con quello di scrittura per reperire il corretto frame (l'ultimo scritto in memoria). Infine è necessario che entrambi i moduli non scrivano/leggano un byte alla volta ma che sfruttino la potenza dei trasferimenti burst.

La struttura realizzata è dunque composta da:

- **AXIS_TO_DDR_WRITER** Questo modulo prende in ingresso un AXI-Stream a 8 bit, bufferizza un numero limitato di pixel in BRAM e quando il buffer è pieno scrive in DDR attraverso una porta AXI-Master il contenuto del buffer.
- **DDR_TO_AXIS_READER** Questo modulo riempe un buffer interno leggendo dalla DDR attraverso una porta AXI-Master e manda in output su una porta AXI-Stream il contenuto del buffer un valore alla volta.

Entrambi i moduli hanno bisogno di un indirizzo di partenza da cui cominciare a scrivere/leggere, mentre per poter lavorare sul corretto frame buffer

è stato inserito una variabile frame_index. Il frame_index varia da 0 a numero_frame_buffer - 1 e viene incrementato dal writer ogni volta che questo comincia a scrivere una nuova immagine (praticamente indica il frame buffer su cui sta lavorando il writer). Questa variabile è poi mandata in ingresso al reader che prima di leggere un nuovo frame andrà a leggere il frame_index e lo decrementerà di 1. In questo modo il reader andrà sempre a leggere il frame più recente scritto dal writer. La separazione in due moduli e l'uso del frame_index implicano anche il completo disaccoppiamento tra scrittura e lettura per quanto riguarda la velocità:

- Se il writer è molto più lento del reader, quest'ultimo leggerà più volte lo stesso frame (senza bloccarsi).
- Se il reader è molto più lento del writer, il reader, che legge sempre il frame più recente, salterà alcuni frame. L'importante è garantire che velocità_scrittura / velocità_lettura > numero_frame_buffer altrimenti il writer potrebbe sovrascrivere il frame che il reader sta correntemente leggendo. Nei nostri casi writer e reader hanno sempre avuto velocità comparabili (circa 30 fps).

Entrambi i moduli utilizzano un AXI-Master a 64 bit nella comunicazione con la DDR per garantire la massima velocità di trasferimento. Le frequenza di clock a cui sono stati testati varia da 100 MHz a 200 MHz.

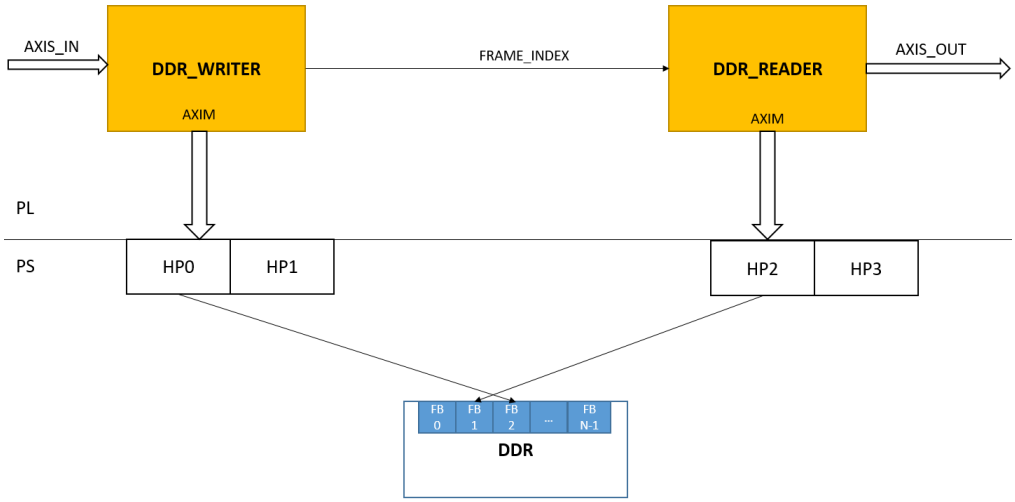


Figure 9: Schema del funzionamento dei moduli axis_to_ddr_writer e ddr_to_axis_reader

2.1 Customizzazione dei moduli via AXI-Lite

Nella versione finale realizzata di entrambi i moduli tutti i parametri (eccetto la width dei dati in ingresso e la dimensione del buffer interno che necessariamente devono essere sintetizzati in hardware) sono modificabili tramite AXI-Lite. Collegando la porta GP dello Zynq a entrambi i moduli sono stati quindi creati dei driver custom per lavorare con reader e writer che consentono di impostare:

- base_address: Indirizzo di partenza
- frame_buffer_dim: Dimensione di ogni frame_buffer in byte (nel nostro caso 640*480 = 307200)
- frame_buffer_num: Numero di frame buffer (fino a un massimo di 255)
- frame_buffer_offset: Offset in byte tra un frame_buffer e l'altro. Questo valore deve essere almeno pari a frame_buffer_dim; in particolare se frame_buffer_offset = frame_buffer_dim tutti i frame_buffer saranno contigui.

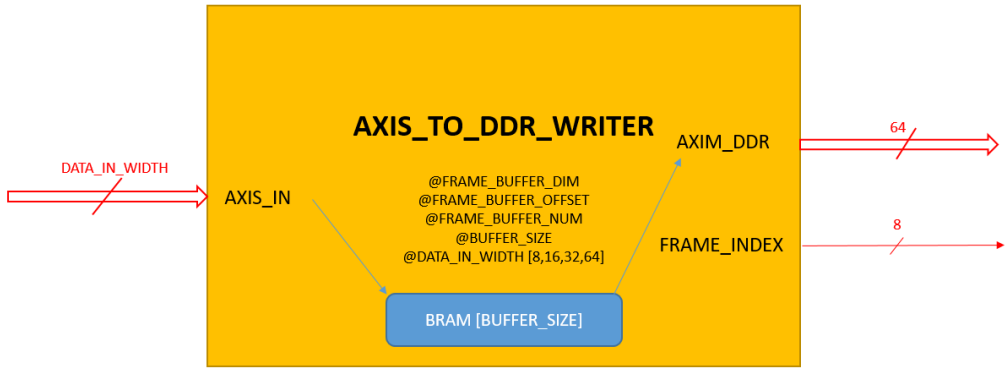


Figure 10: Interfaccia di I/O del modulo axis_to_ddr_writer

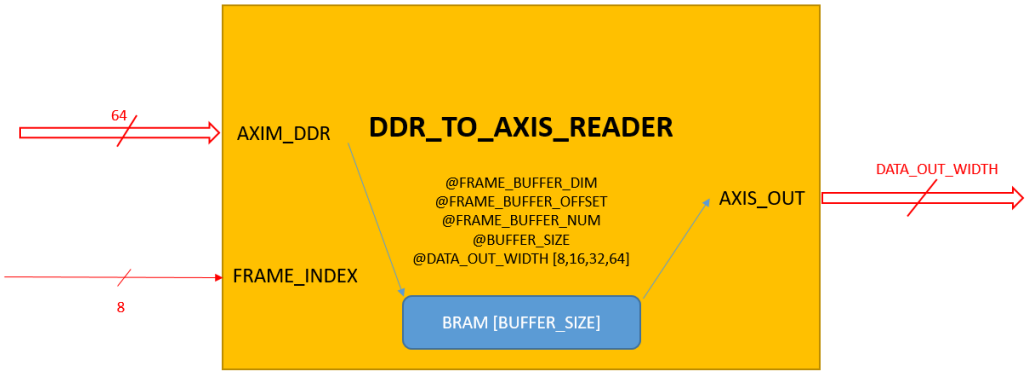


Figure 11: Interfaccia di I/O del modulo ddr_to_axis_reader

2.2 Test e introduzione del Pattern Generator

I primi test hanno riguardato l'uso dei moduli writer e reader separatamente, inzialmente senza frame_index solamente per testare che le scritture e le letture avvenissero correttamente. Una volta ultimati, è stato aggiunto un pattern generator il cui scopo è quello di simulare un sensore. Questo pattern generator, realizzato anch'esso in HLS, genera una croce che rimbalza quando incontra i bordi dello schermo. Infine è stato collegato il reader al modulo di interfacciamento con la VGA. Per fare comunicare il reader con

clock a 100 MHz con la VGA con clock a 25.175 MHz è stata utilizzata una fifo asincrona che permette di fare di ponte tra i due diversi domini di clock. Allo stesso modo è stato collegato un clock a 24 MHz al pattern generator per simulare la frequenza possibile di un sensore ed è stata inserita una fifo tra la regione del sensore (e in futuro di tutta la pipeline) e il writer, che ha in ingresso un clock a 100 Mhz.

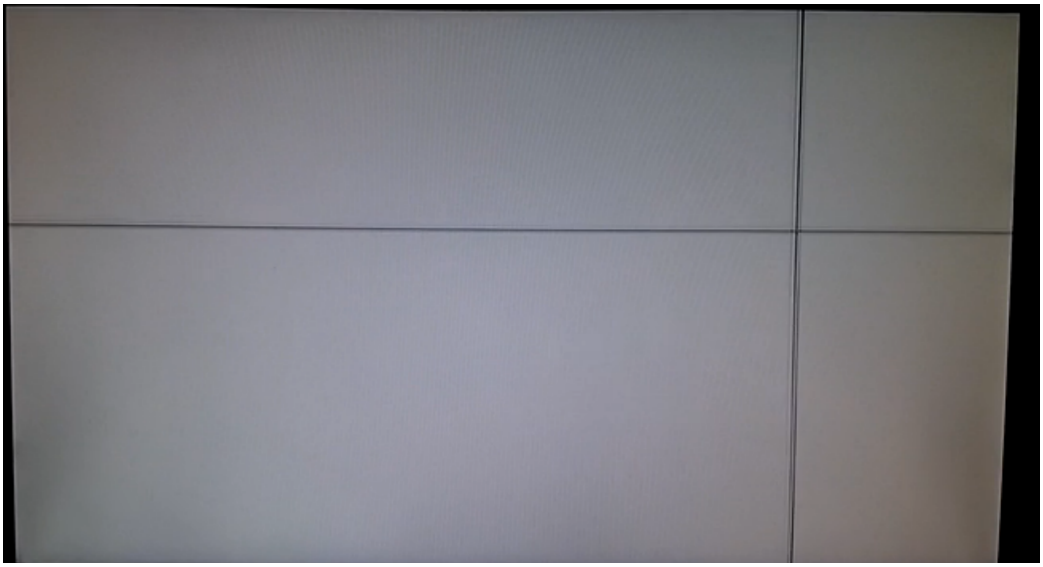


Figure 12: Immagine del pattern generator su VGA con trasferimento immagini in DDR

2.3 Introduzione dell'Interrupt inviato al PS

Per permettere ad ARM di lavorare in maniera veloce ed efficiente con le immagini generate si è scelto di inviare un interrupt al Processing System ogni volta che il modulo di scrittura ha finito di generare un'immagine. Una volta ricevuto l'interrupt il programma su ARM non deve far altro che reperire il valore del frame_index corretto per poter ottenere il numero dell'ultimo frame scritto in memoria.

Per aggiungere la generazione dell'interrupt è stato sufficiente aggiungere un modulo AXI GPIO di input che riceve in ingresso il frame_index: ogni

volta che questo cambia, il modulo genera automaticamente un interrupt; questo interrupt è stato poi collegato al modulo Zynq Processing System.

Su Xilinx SDK è stato inoltre necessario programmare la parte PS attraverso opportune funzioni per abilitare l'interrupt; il tutto è stato gestito con una semplice libreria che permette di registrare l'interrupt nell'Interrupt Controller di ARM e di registrare una funzione di callback che riceverà in ingresso il valore del frame_index. La funzione di callback sarà quindi invocata ad ogni generazione di un frame e consentirà di interagire immediatamente con il frame generato (ad esempio per inviarlo via ethernet ad un host).

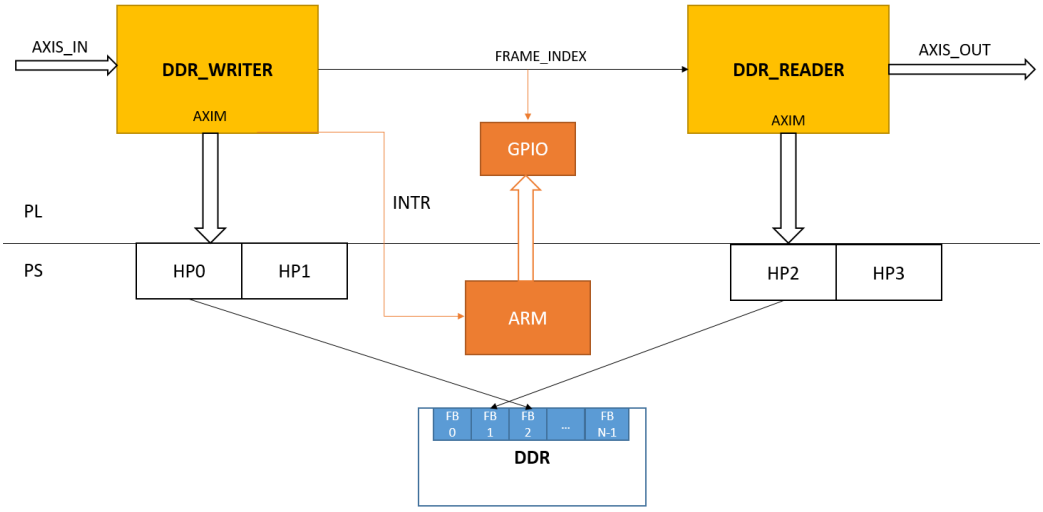


Figure 13: Architettura per il trasferimento di immagini in DDR con interrupt

3 Interfacciamento dei sensori all'architettura

Una volta realizzata l'infrastruttura di scrittura e lettura delle immagini in memoria sono stati collegati dei sensori reali allo Zynq. Per realizzare questa connessione sono state introdotte due parti a monte dell'infrastruttura già realizzata: la prima parte è device-dependent e si occupa di interfacciarsi con i segnali generati dallo specifico sensore per generare i segnali canonici LINE_VALID e FRAME_VALID. La seconda parte, indipendente dallo specifico sensore, si occupa di trasformare i precedenti segnali in un AXI Stream. La parte compresa tra l'axi stream così generato e il modulo di scrittura in memoria è composto da filtri; la peculiarità del protocollo axi stream combinato coi filtri è che è posibile inserire, rimuove, abilitare e disabilitare i filtri in maniera veloce e totalmente indipendente dai moduli che si trovano a valle o a monte della pipeline.

3.1 Modulo LF_VALID_TO_AXIS

Il compito di questo modulo è di prendere in ingresso i segnali standard FRAME_VALID, LINE_VALID e DATA e trasformarli in un AXI Stream. L'implementazione del modulo è molto semplice: scrive sullo stream di output solamente quando FRAME_VALID e LINE_VALID sono entrambi a 1 e gestisce coerentemente il reset, aspettando un fronte di discesa di FRAME_VALID in modo da ignorare l'immagine parziale in arrivo al momento del reset. Si noti che il modulo non ha bisogno di conoscere la dimensione del frame in ingresso.

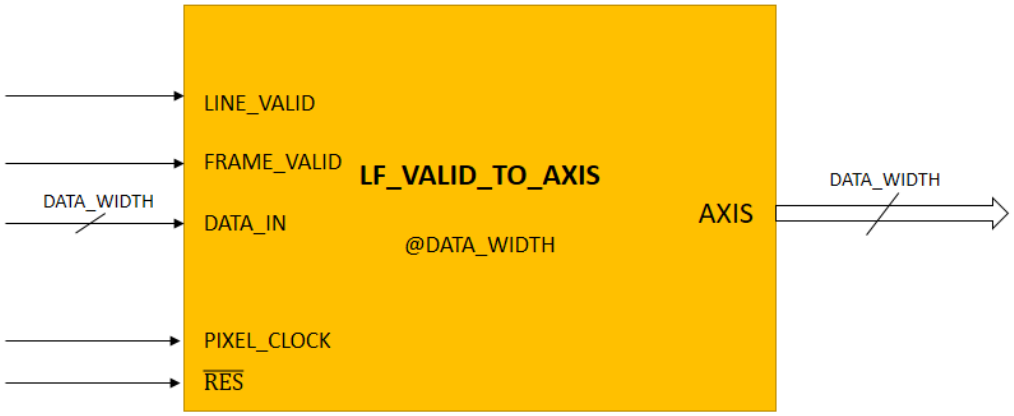


Figure 14: Interfaccia di I/O del modulo LF_VALID_TO_AXIS

3.2 Interfacciamento del sensore OV7670

Il modulo di interfacciamento diretto con il sensore OV, chiamato OV7670_Interface, era già stato realizzato in Vivado HLS. Il modulo inizialmente presente si occupava di gestire i segnali di controllo in ingresso provenienti dall'OV, di campionare i valori grayscale e di restituirne i 4 bit più significativi (essendo la VGA sulla ZedBoard a 4 bit). In questo caso sono stati separati i compiti svolti dal modulo ed è stato eliminato il vincolo della VGA a 4 bit; i moduli così modificati sono i seguenti:

- **OV7670_Interface** Il modulo che si interfaccia direttamente con i segnali VSYNC e HREF generati dall'OV e campiona in maniera coerente i dati, generando i segnali canonici LINE_VALID e FRAME_VALID. Avendo questa sola resonsabilità, questo modulo non avrà mai bisogno di modifiche qualora venga modificato il comportamento dell'OV attraverso un'opportuna programmazione.
- **OV7670_Grayscale** Il modulo che filtra i dati in arrivo dal sensore OV è diventato un semplice filtro AXI Stream, il cui compito è quello di filtrare i byte di indice dispari in ingresso. Alla fine di questo modulo abbiamo quindi il flusso di immagini in grayscale generato dall'OV7670.

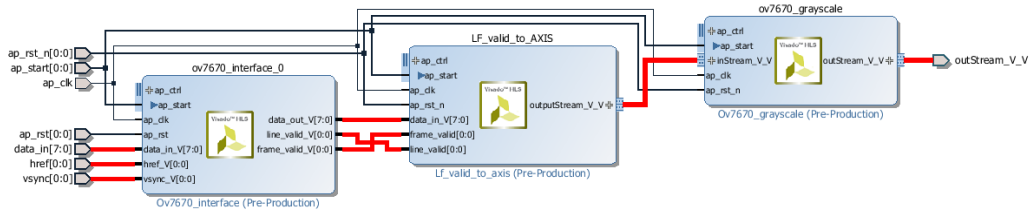


Figure 15: Schematico della pipeline di elaborazione dell'OV7670

Una volta realizzati i moduli è stato sufficiente collegare l'uscita del modulo OV7670_Grayscale al modulo axis_to_ddr_writer per ottenere in uscita sulla VGA le immagini generate dal sensore. E' stato inoltre inserito un modulo già realizzato in HLS che permette di applicare vari filtri di convoluzione all'immagine in ingresso (smooth, edge detection, ...).

Figure 16: Immagine dell'OV7670 su VGA con trasferimento in DDR delle immagini

3.3 Interfacciamento dei sensori stereo Aptina

Anche il modulo di interfacciamento con i sensori Aptina, scritto in VHDL, era già stato realizzato. Questo componente, chiamato SerDes, si occupa appunto della deserializzazione dei dati LVDS in ingresso e della generazione del clock originario (si ricorda che il clock del flusso serializzato è 18 * clock_originario). Il SerDes genera in uscita i pixel left e right, i corrispondenti segnali line_valid e frame_valid, il clock ricostruito e altri segnali di controllo che vengono messi a zero quando il SerDes incontra un problema nella ricostruzione dell'immagine (pixel disallineati, segnali di controllo ripetuti o in ordine diverso da quello aspettato,...).

Per codificare i segnali di controllo di start_of_frame (SOF), end_of_frame (EOF), start_of_line (SOL) e end_of_line (EOL) erano stati eliminati 4 valori dai possibili valori trasmessi dai sensori Aptina (corrispondenti ai byte 0,1,2,3). Il SerDes presentava però un bug nella ricostruzione dell'immagine nel momento in cui i sensori catturavano un pixel con valore tra 0 e 3. Il problema è stato risolto realizzando un nuovo front-end in VHDL, costituito da una macchina a stati, il cui compito è quello di passare dalla forma SOF-SOL-EOL-EOF alla forma LINE_VALID e FRAME_VALID.

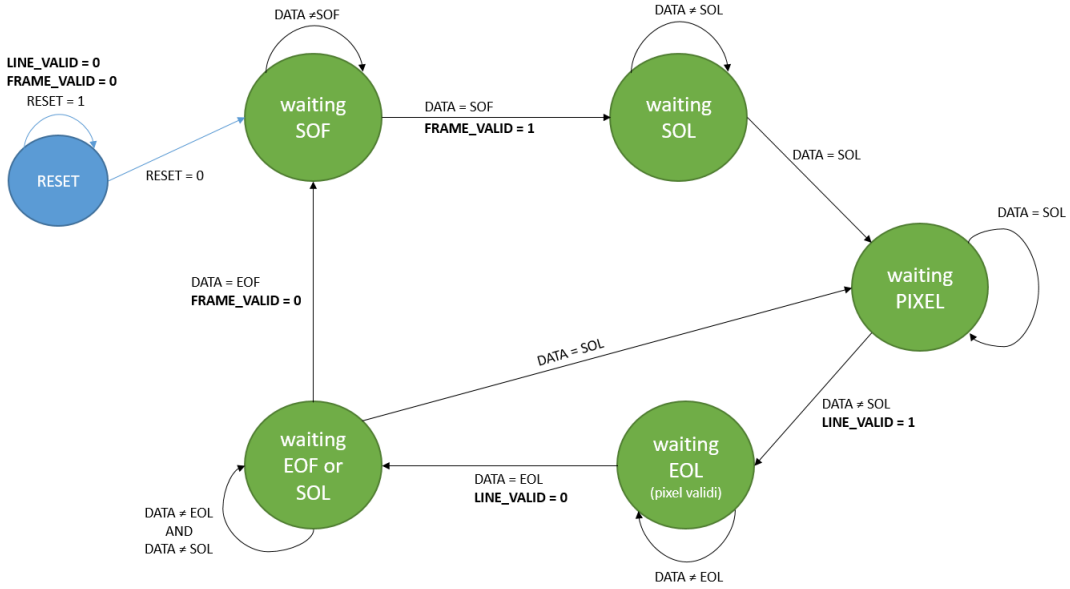


Figure 17: Macchina a stati finiti del front-end del SerDes

Per collegare il SerDes al resto della pipeline è stato sufficiente aggiungere dopo di esso due moduli LF_VALID_TO_AXIS e utilizzare due moduli axis_to_ddr_writer.

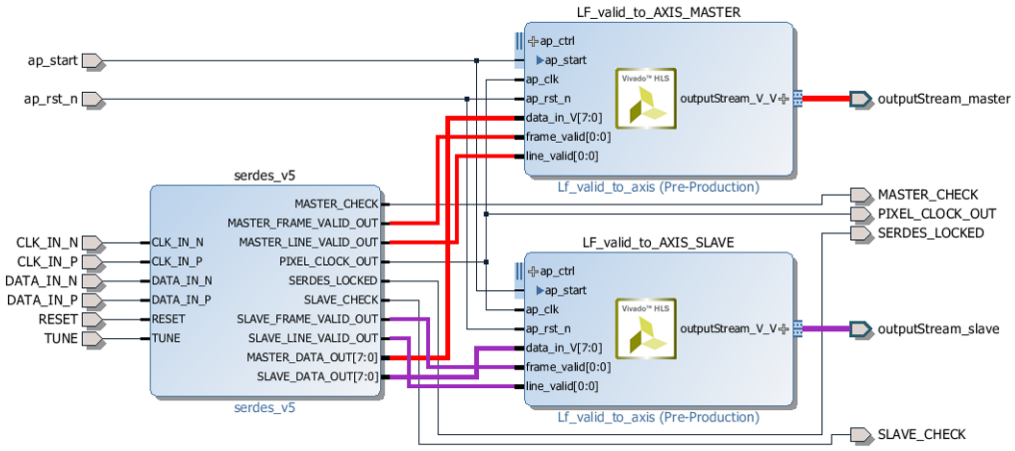


Figure 18: Pipeline base di elaborazione dei sensori stereo Aptina

Una volta generato il bitstream è necessario programmare i sensori via I2C. Il codice C da eseguire su ARM era già fornito.

Figure 19: Immagini stereo a 8 bit ottenute attraverso trasferimento via Ethernet (si veda sezione 4)

4 Porting del tool di visualizzazione delle immagini su Windows

Per superare le limitazioni sulla qualità dell'immagine imposte dalla VGA a 4 bit presente sulla ZedBoard, è stato realizzato un porting per Windows del programma che consente ad un host di leggere le immagini in arrivo dalla scheda attraverso una connessione Ethernet punto-punto (senza passare attraverso un router). Il porting del programma, precedentemente scritto per sistemi Linux in linguaggio nativo C da Simone Mingarelli, è stato altamente facilitato dall'uso delle librerie di sistema WinSock. Questa libreria consente di accedere in maniera facilitata ai protocolli TCP/IP ed espone al programmatore delle primitive (quali socket, bind, connect) del tutto analoghe a quelle dei sistemi Unix. Per utilizzare il medesimo codice scritto per Unix è stato sufficiente introdurre una parte preliminare di inizializzazione legata alla libreria WinSock.

Per rendere possibile la visualizzazione corretta delle immagini è stato necessario disabilitare il firewall di Windows e impostare una dimensione del pacchetto dati superiore rispetto a quella usata su Linux (32kB). Inoltre, nella versione sviluppata per ricevere le immagini dei sensori Aptina (in cui la banda necessaria è doppia rispetto a quella per trasmetere le immagini dell'OV7670), è stato necessario aumentare il buffer di ricezione di sistema fino a valori piuttosto elevati (64k).

L'applicativo sfrutta il protocollo di rete UDP e utilizza un pacchetto che contiene soltanto i dati dell'immagine e un indice [0, DIM_IMMAGINE / DIM_PACCHETTO -1] per contare il numero del pacchetto all'interno di un frame. Il comportamento dell'applicazione è del tipo "tutto o niente": ci si aspetta di ricevere tutti i pacchetti relativi a un frame e nello stesso ordine in cui sono stati inviati. Nel caso in cui queste condizioni non siano verificate, i pacchetti precedentemente ricevuti vengono scartati e ci si pone nuovamente in attesa del primo pacchetto di un frame; in caso positivo l'immagine viene invece visualizza a video tramite le librerie OpenCV.

E' stata infine aggiunta una funzionalità che permette, alla pressione di un tasto, di salvare l'ultima immagine ricevuta (o l'ultima coppia di immagini ricevute nel caso dei sensori Aptina) su file. Questa funzione utilizza attual-

mente funzioni windows-dependent: non è possibile infatti usare la classica getchar perché nel caso in cui non sia stato premuto alcun tasto l'applicazione deve continuare a ricevere le immagini; il comportamento realizzato è quello di una lettura non bloccante da standard input.

Part IV
ALGORITMI DI VISIONE STEREO

Una volta costruita l'architettura hardware e software in grado di salvare e visualizzare immagini stereo, si è passati alla progettazione hardware di un sistema di visione stereo che ha portato alla generazione delle mappe di disparità. Si procede con una breve introduzione (non esaustiva) alla stereo visione e agli algoritmi utilizzati fino ad arrivare alla specifica implementazione e alla realizzazione su architettura Zynq.

5 Introduzione agli algoritmi di visione stereo

Come già sottolineato, la visione stereo è in grado di generare mappe di disparità attraverso la corrispondenza degli oggetti, o dei punti, tra immagini corrispondenti (correspondence problem); il problema della corrispondenza richiede operazioni computazionalmente onerose ed è pertanto portato all'implementazione su FPGA. Un modo per affrontare questo problema è scomporlo in due fasi, che verranno analizzate di seguito: rettificazione e stereo matching.

5.1 Rettificazione

In linea di principio, dato un punto A in un'immagine, per trovare il punto corrispondente B nella seconda immagine è necessario effettuare una ricerca in due dimensioni lungo tutta la seconda immagine. Per semplificare la ricerca si procede inizialmente con la fase di rettificazione. Questo processo consente di rimuovere la distorsione delle lenti e di modificare l'immagine in ingresso in maniera tale che punti corrispondenti si trovino sulla stessa linea retta orizzontale. Geometricamente parlando, lo scopo della rettificazione è quello di rendere le due camere virtualmente parallele e perfettamente allineate. La rettificazione costituisce quindi una fase preliminare che consente di ridurre drasticamente il costo dello stereo matching sia in termini di risorse che di tempi.

Il modulo che si occuperà di effettuare la rettificazione può quindi essere visto come un filtro che, date in ingresso le immagini distorte, produce in uscita le immagini rettificate sulle quali poi effettuare il matching dei punti corrispondenti.

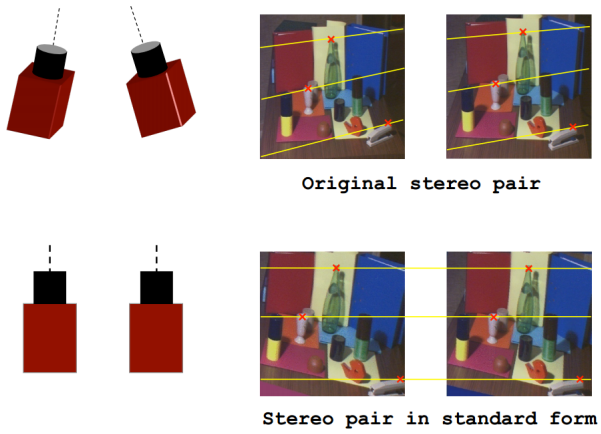


Figure 20: Rettificazione

Affinché sia possibile effettuare la rettificazione real-time, è necessario effettuare una fase preliminare offline, chiamata *calibrazione*, che consente di ricavare i dati necessari a rettificare corretamente le immagini in ingresso. Software come OpenCV e Matlab offrono supporto alla calibrazione e consentono di generare le tabelle di correzione (*displacement tables*) che per ogni punto dell'immagine indicano dove si trova il punto dell'immagine non distorta che è necessario mandare in output (la tabelle possono essere in forma assoluta o in forma relativa; in questa seconda forma viene indicato l'offset positivo o negativo rispetto al punto corrente).

Il processo di rettificazione è una componente fondamentale per ottenere una corrispondenza stereo di qualità. Se la calibrazione non è effettuata correttamente i dati tridimensionali ricavati saranno inaccurati o addirittura privi di significato.

5.2 Stereo Matching e SGM

Il secondo stadio, una volta superata la fase di rettificazione, si occupa di trovare per ogni punto dell'immagine di riferimento (tipicamente la left) il

punto corrispondente nell'immagine target (right). E' bene ripetere che grazie alla rettificazione è possibile effettuare questa ricerca all'interno della stessa riga. Una volta trovata la corrispondenza diventa facilmente ricavabile la mappa di disparità; ad ogni punto dell'immagine di riferimento è associata un valore di intensità (la disparità) che indica la distanza del punto corrispondente: quanto più i punti corrispondenti sono lontani e tanto più saranno vicini alla camera (e quindi avranno una disparità alta); viceversa punti corrispondenti molto vicini faranno riferimento ad un punto lontano dalla camera e quindi la disparità ad essa associati sarà bassa. Per limitare le risorse e aumentare le prestazioni la disparità non viene calcolata per tutti i valori fisicamente possibili [0,inf], ma all'interno di un intervallo più ristretto [DISPARITY_MIN, DISPARITY_MAX]. Da quanto detto sopra si può dedurre che al diminuire di DISPARITY_MIN l'algoritmo sarà in grado di calcolare correttamente la disparità per punti sempre più lontani, mentre all'aumentare di DISPARITY_MAX verranno rilevati correttamente punti più vicini alla camera.

La scelta dell'algoritmo di matching è fondamentale ed influenza ogni aspetto della progettazione, tra cui le prestazioni, le risorse hardware utilizzate e la qualità dalla mappa generata. Come sempre è necessario effettuare un compromesso che permetta di soddisfare fin dove possibile tutti questi fattori. Esistono due categorie di algoritmi stereo:

- **Algoritmi locali**: la scelta del punto corrispondente è effettuata considerando solo le vicinanze del punto in esame ed è di tipo "Winner Takes All"; l'algoritmo più semplice possibile è quello che, preso un punto nell'immagine di riferimento, effettua il confronto in N punti dell'immagine target (nella stessa riga) e sceglie quello che minimizza la differenza in valore assoluto (cioè quello più simile al punto di riferimento). Altri algoritmi poco più evoluti effettuano la ricerca in una finestra di matching aggregando i costi trovati. Gli algoritmi locali sono prevalentemente performanti e utilizzano poche risorse, però in alcuni casi la mappa di disparità generata può non essere di buona qualità, soprattutto in regioni uniformi (in cui la sola analisi locale non basta per ricavare informazioni). La scelta di aggregare i costi di una finestra porta inoltre ad ottenere un'ulteriore perdita di qualità in prossimità dei bordi, in quanto punti appartenenti a regioni di spazio ben distinte vengono utilizzati assieme.

- **Algoritmi globali**: la scelta del punto corrispondente è effettuata con-

siderando tutta l'immagine; in particolare viene cercato un punto che minimizzi una determinata funzione costo (chiamata energia) lungo tutta l'immagine. Esiste poi una categoria intermedia di algoritmi, detti **semi-globali**, in qui la funzione costo deve essere minimizzata all'interno di una porzione dell'immagine (ad esempio lungo determinati percorsi). Tale funzione è costo è spesso espressa come $E(d) = E_{data}(d) + E_{smooth}(d)$, dove E_{data} misura la corrispondenza diretta tra i punti in esame (spesso implementata come una semplice funzione costo tra pixel) e E_{smooth} è un termine che penalizza le variazioni di disparità. E' possibile ricavare direttamente le seguenti osservazioni basate sulla definizione di E_{smooth}:

- In regioni uniformi il termine E_{smooth} sarà sufficientemente alto da impedire variazioni considerevoli di disparità, consentendo quindi di ridurre notevolmente il rumore.
- Nelle regioni di discontinuità (bordi) vi sarà una fase di transitorio in cui inizialmente E_{smooth} impedirà la variazione di disparità. Nel momento in cui tale variazione si paleserà sempre più, entrambi i termini verranno ridotti e consentiranno di effettuare il cambio di disparità.

L'algoritmo utilizzato nel nostro caso è *SGM*, un algoritmo semi-globale che effettua il calcolo dell'energia lungo 8 percorsi, chiamati *scanline*, a 45° l'uno dall'altro (le scanline verranno numerate da 0 a 7 in senso orario a partire dalla scanline a sinistra/ovest). Inoltre le scanline e le relative funzioni costo sono indipendenti fra di loro.

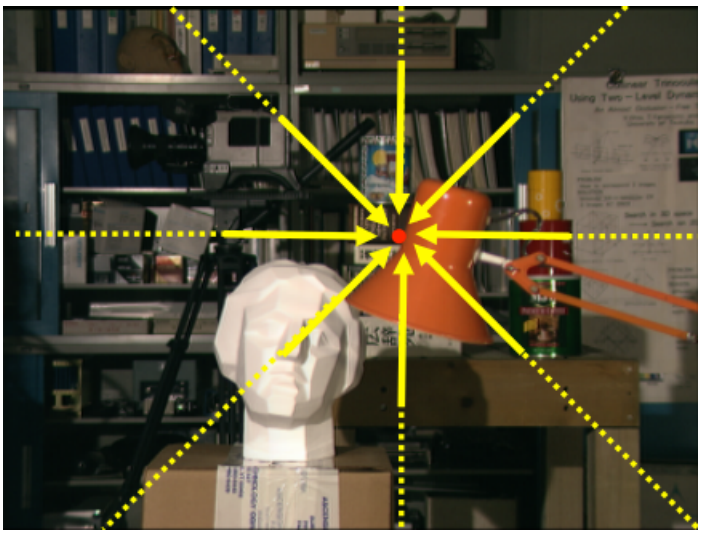


Figure 21: Le 8 scanline di SGM

La funzione costo è calcolata lungo gli 8 percorsi all'interno dell'intervallo di disparità $d \in [DISPARITY\_MIN, DISPARITY\_MAX]$. Indicando con x e y le coordinate del punto analizzato, la formula della funzione costo è la seguente (qui è riportata la formula per la scanline 0):

$$L(x,y,d) = C(x,y,d) + min\{L(x-1,y,d), L(x-1,y,d-1) + P1, L(x-1,y,d+1) + P1, L(x-1,y,i) + P2\}$$

Come si può notare la funzione costo è ben suddivisa nelle due componenti E_{data} ed $E_{smoooth}$; $E_{data} = C(x,y,d)$ è il risultato di una funzione costo locale applicata ai punti analizzati e che è indipendente dal resto dall'algoritmo (è quindi possibile applicare funzioni costo locali differenti ottenendo output di maggiore o minore qualità); il termine $C(x,y,d)$ è inoltre indipendente dalla specifica scanline in quanto calcolata sempre negli stessi punti. Il termine di smoothness, realizzato in maniera ricorsiva, consente invece di tenere traccia della storia passata lungo lo specifico percorso e tende a penalizzare i cambi di disparità attraverso l'utilizzo di due pesi P1 e P2 (con P1 $<$ P2): il cambio di disparità si verifica quando un punto immediatamente a destra o a sinistra riesce a vincere una piccola penalità (P1), oppure quando un punto più lontano riesce a vincere una grande penalità (P2). Per evitare il tendere all'infinito della funzione costo viene infine sottratto al risultato trovato il termine $minL(x-1,y,i)$.

Una volta calcolate le funzioni costo lungo le 8 scanline, queste vengono aggregate e ne viene infine calcolato il minimo. L'indice d del minimo rappresenta la vera disparità calcolata da SGM.

SGM si presta molto bene all'implementazione su FPGA in quanto composto soltanto da operazioni tra interi e produce una mappa di disparità di qualità tipicamente superiore rispetto agli algoritmi locali. Nonostante ciò utilizza per sua natura molte risorse di calcolo (dipendenti linearmente dalla larghezza dell'intervallo di disparità) e di memoria; la fase di ottimizzazione dell'algoritmo sarà quindi cruciale per poter rendere SGM implementabile in hardware.

6 Implementazione della Rettificazione

Il primo modulo da realizzare si deve occupare della rettificazione di una immagine generata dalla camera; in prima battuta si può dire quindi che l'interfaccia dell'IP avrà due axi-stream a 8 bit, uno di input e uno di output. Affinché sia possibile rettificare l' immagine è però necessario poter inviare al modulo le tabelle di correzione ed altri eventuali parametri: sarà quindi necessario introdurre una parte di controllo che permetta la comunicazione tra ARM e il modulo realizzato.

Realizzare la rettificazione in hardware richiede potenzialmente molte risorse, sia per quanto riguarda le LUT e i Flip Flop (utilizzati per i calcoli e per costruire lo scheletro della pipeline interna), sia per quanto riguarda le BRAM. In particolare gli oggetti che necessitano di utilizzare le BRAM sono due:

- Le tabelle di rettificazione, che per ogni pixel dell'immagine contengono lo scostamento, positivo o negativo, per trovare il pixel corretto
- Il line buffer, un array bidimensionale grande N x IMG_WIDTH che contiene N righe di pixel precedentemente letti. Ovviamente nel momento in cui si trova la posizione del pixel corretto è necessario che tale pixel sia già stato letto (ovvero che la camera non debba ancora generarlo) e che sia stato memorizzato; il line buffer si occupa proprio di bufferizzare un numero limitato di righe affinchè sia possibile accedere ai pixel necessari a rettificare l'immagine.

  La dimensione del line buffer influenza notevolmente il numero di BRAM utilizzate e rappresenta senza dubbio il parametro sui cui porre maggiore attenzione e su cui effettuare le maggiori ottimizzazioni. E' importante sottolineare come questa dimensione dipenda da vari fattori, tra cui:

  - La risoluzione dell'immagine: al crescere della risoluzione aumenta il numero di linee necessarie.
  - La distorsione delle lenti: più le lenti sono distorte e più sono necessarie linee in quanto bisogna spostarsi di più all'interno dell'immagine.
  - La posizione reciproca delle camere: più i sensori sono allineati e minore sarà lo "sforzo" necessario a rettificare le immagini (e quindi il buffer richiesto sarà più piccolo).

6.1 Algoritmo ottimale per il calcolo del numero minimo di linee del line buffer

Il numero di linee per il line buffer è direttamente ricavabile dalle tabelle di rettificazione in forma relativa. In una precedente versione di questo modulo il numero di linee per il line buffer era calcolato come scostamento_positivo_max - scostamento_negativo_min: se per un punto lo scostamento verticale positivo massimo è di 40 righe (cioè il punto corrispondente di trova 40 righe più sotto) mentre per un altro punto lo scostamente verticale negativo minimo è di -30 righe (cioè il punto corrispondente si trova 30 righe più sopra), allora il numero necessario di righe per il line buffer è 70.

Il calcolo evidenziato sopra, sebbene corretto, nasconde un approccio di tipo "worst-case": osservando le tabelle di rettificazione si nota infatti come il valore scostamento_positivo_max si raggiunga tipicamente nella parte alta dell'immagine (all'inizio c'è bisogno di spostarsi molte linee sotto), mentre il valore scostamento_negativo_min viene raggiunto nella parte bassa dell'immagine (alla fine c'è bisogno di spostarsi molte linee sopra). Il passaggio dal valore massimo al valore minimo è lento e graduale lungo tutta l'altezza dell'immagine. Da queste osservazioni è quindi lecito ipotizzare che anche un valore inferiore alla loro differenza sia sufficiente per effettuare correttamente la rettificazione, in quanto non accadrà mai di doversi spostare contemporaneamente di scostamento_positivo_max e di scostamento_negativo_min righe.

L'algoritmo realizzato è fondato su queste ipotesi e, partendo dal numero di linee trovato nel caso peggiore, verifica che sia possibile realizzare la rettificazione ad ogni riga, ovvero se per ogni riga il line buffer contiene le righe corrispondenti a scostamento_max e scostamento_min per quella riga. Se ciò si verifica per tutte le righe, allora è possibile decrementare di uno il numero di linee ed effettuare lo stesso controllo. L'algoritmo si ferma nel momento in cui per una riga non è possibile trovare nel line buffer tutte le righe di cui ha bisogno. Intuitivamente ciò si verifica quando il line buffer è così piccolo che una riga utile alla rettificazione viene sostituita dalla riga in ingresso dai sensori che sta venendo salvata. Si può pensare al processo di rettificazione come ad un inseguimento tra la riga corrispondente a scostamento_min e la riga in ingresso: nel momento in cui esse si sovrappongono non è più possibile rettificare correttamente l'immagine.

L'algoritmo descritto sopra è esplicitato nel diagramma seguente; row e current_line sono rispettivamente la riga su cui si sta effettuando la rettificazione e la linea che sta venendo letta dai sensori. prima_riga(r) e ultima_riga(r) rappresentano le righe corrispondenti a scostamento_min e scostamento_max per la riga r. Last_used_row rappresenta l'ultima riga necessaria per effettuare la rettificazione: se infatti all'inizio dell'immagine ci si sposta molto in basso e alla fine dell'immagine ci si sposta molto in alto, ci saranno righe in alto e in basso che non saranno utili ai fini della rettificazione e che scompariranno dall'immagine in uscita. Utilizzando last_used_row è possibile non salvare le ultime righe dell'immagine non utilizzate e in qualche modo "rallentare" l'inseguimento tra la riga rettificata e quella in ingresso. Ciò permette quindi di risparmiare ulteriori risorse.

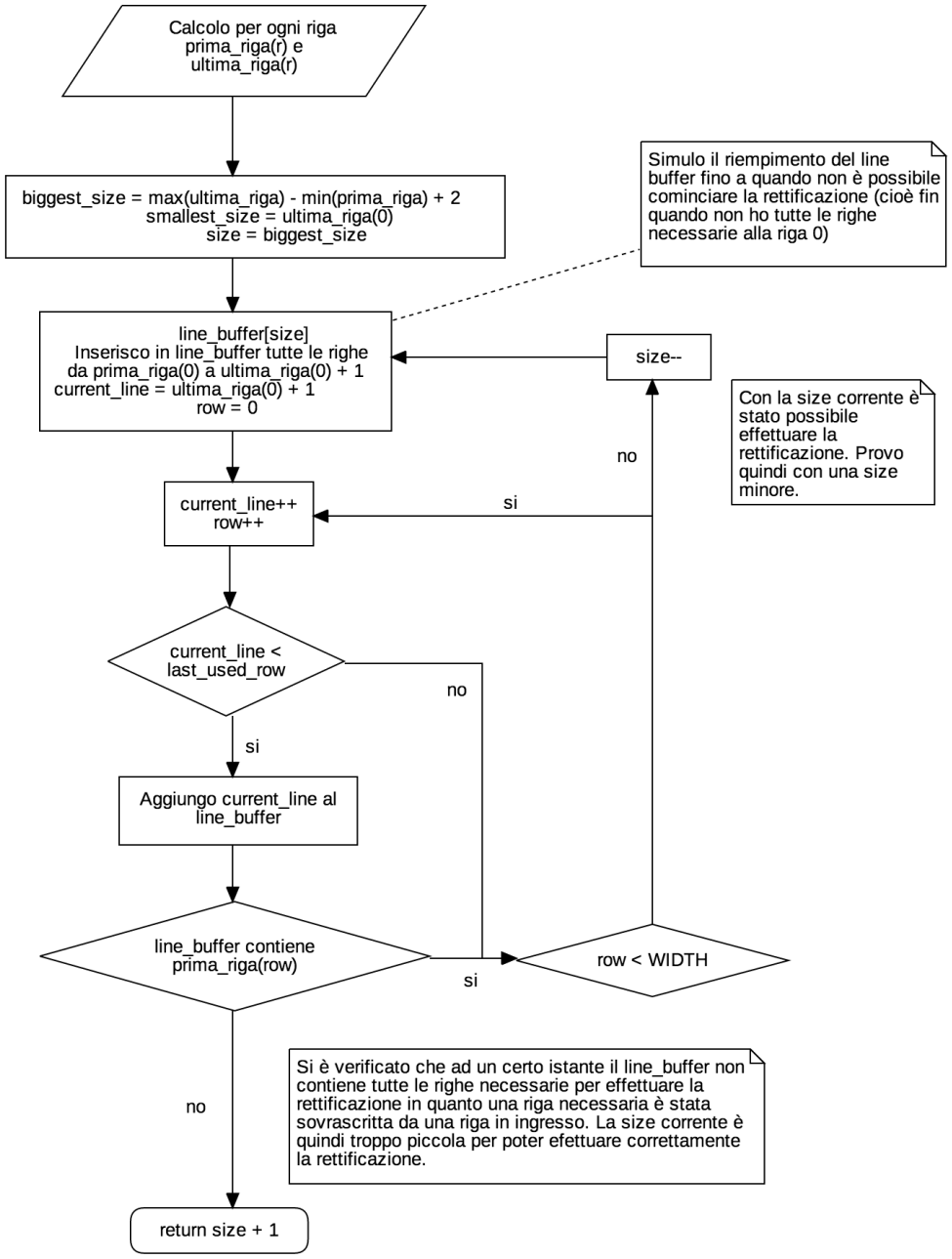


Figure 22: Diagramma di flusso dell'algoritmo per il calcolo del numero minimo di righe per il line buffer nel processo di rettificazione

Per non stallare la pipeline a valle è necessario che punti corrispondenti vengano scritti in output contemporaneamente dai due moduli di rettificazione. Ciò implica che la riga in cui comincia la rettificazione (chiamata *first_rectification_row* sia la stessa per entrambi i moduli. Una volta eseguito l'algoritmo descritto sopra, sarà quindi necessario verificare che il sensore con first_rectification_row minore possa effettuare la rettificazione con un ritardo dovuto alla rettificazione dell'altra immagine. Ciò in generale non è verificato e per permettere la rettificazione contemporanea sarà necessario aggiungere poche (ma inevitabili) linee di line buffer al modulo con first_rectification_row minore, in modo da far fronte al ritardo aggiuntivo.

Volendo infine realizzare un modulo indipendente dalla camera left o right, verrà scelto il numero di linee più alto trovato tra quello per l'immagine left e quello per l'immagine right. In questo modo, pena un minimo overhead di risorse, si può evitare di dover gestire due moduli separati per i due sensori e di dover poi effettuare due sintesi diverse.

6.2 Realizzazione del modulo e integrazione con la pipeline

Oltre all'ottimizzazione sulla dimensione del line buffer sono state applicate ulteriori ottimizzazioni già presenti in precedenti versioni del modulo:

- Le tabelle memorizzate in BRAM non sono complete ma sono ridotte, cioè sotto-campionate. Il vantaggio è che ciò consente di utilizzare meno memoria (nella nostra implementazione viene salvato uno scostamento ogni 16 sia sulle righe che sulle colonne, quindi la tabelle sono 256 volte più piccole), lo svantaggio è che è necessaria una fase preliminare di ricostruzione del valore dello scostamento a partire da quelli noti tramite interpolazione. Dal fattore di scaling e dal numero di bit utilizzati per memorizzare gli scostamenti dipende in maniera diretta la qualità e la correttezza dell'immagine di uscita.

- Le tabelle non sono memorizzate come valori floating-point ma sono moltiplicate per un determinato fattore e discretizzate. Ciò consente di risparmiare numerose unità di calcolo (LUT) in quanto le operazioni floating-point sono notevolmente più pesanti in termini di risorse e di performance. Ancora una volta il fattore di discretizzazione influenza la qualità finale dell'immagine rettificata.

Per quanto riguarda la parte di controllo, in base all'algoritmo di calcolo delle linee del line buffer è necessario che il modulo conosca, oltre alle tabelle:

- last_used_row, che rappresenta l'ultima riga utile per generare l'immagine di uscita
- first_rectification_row, che rappresenta la prima riga in cui è possibile cominciare il processo di rettificazione, cioè la prima riga per la quale nel line buffer sono contenute tutte le righe per poter rettificare la riga 0.

L'unico parametro che non è possibile passare come argomento ma che è necessario cablare nel codice è la dimensione del line buffer, in quanto essa influenza la sintesi hardware. Se necessario è comunque possibile inserire un valore più elevato di quello ottimo in modo da realizzare una sintesi più flessibile a possibili modifiche della configurazione dei sensori (nel nostro caso la sintesi è stata fatta con 100 righe di line buffer quando ne sarebbero bastate poco più di 70).

La comunicazione ARM-to-Rectification è stata realizzata attraverso un'opportuna interfaccia Axi-Slave grazie alla quale è possibile passare gli argomenti di configurazione al modulo. In particolare vengono inviati i valori di last_used_row, first_rectification_row, e un terzo parametro chiamato displacement_table_intr che volge il compito di segnalare la volontà dell'ARM di comunicare con il modulo: una volta impostati i nuovi valori viene settato il displacement_table_intr a 1 dalla cpu. Il modulo di rettificazione tra una immagine e l'altra, cioè nella fase di vertical blanking, verifica che tale valore sia a 1 e in caso positivo aggiorna i parametri e riporta il displacement_table_intr a 0.

Per quanto riguarda l'invio delle tabelle, si è osservato che inviare direttamente le tabelle tramite l'interfaccia Axi avrebbe comportato un consumo di risorse troppo elevato. La strategia alternativa utilizzata è quella di utilizzare, come per i moduli di lettura e scrittura delle immagini, un'ulteriore interfaccia di tipo Axi-Master con la quale reperire nella memoria DDR le tabelle. Attraverso l'interfaccia di controllo Axi-Slave quindi la cpu invia anche l'indirizzo in memoria delle tabelle e il modulo stesso, una volta ricevuto l'interrupt, andrà a leggere in memoria le tabelle aggiornate.

Si riporta per chiarezza l'interfaccia finale del modulo di rettificazione e lo pseudo-algoritmo eseguito in hardware:

Figure 23: Interfaccia di I/O del modulo di rettificazione

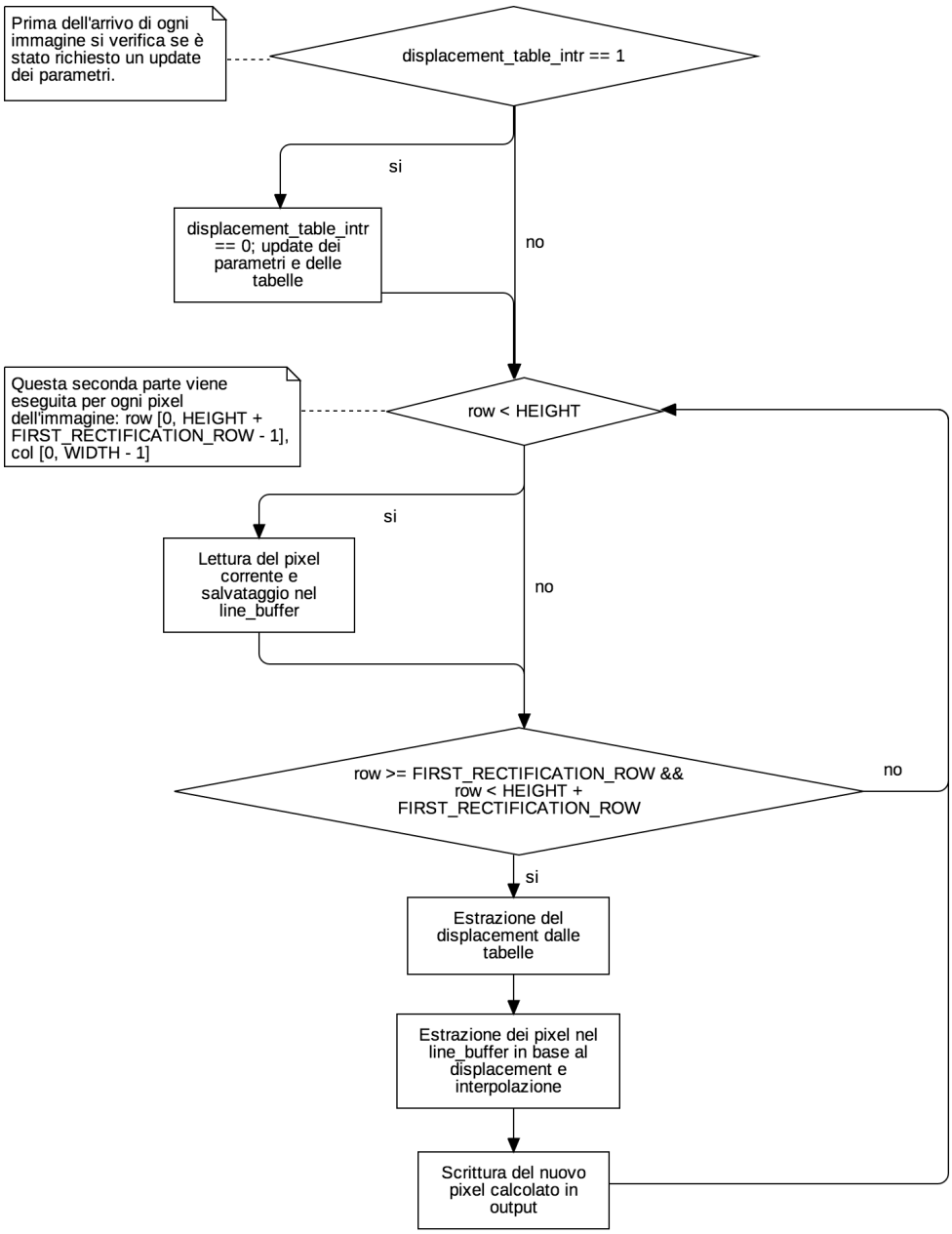


Figure 24: Algoritmo eseguito in hardware dal modulo di rettificazione

Infine vengono riportate le risorse utilizzate dal modulo di rettificazione. Come si può notare i consumi sono sufficientemente contenuti (considerando due moduli istanziati si parla di un quarto delle BRAM e del 20% di LUT) e lasciano ampio margine per poter realizzare e testare il modulo di stereo matching, che sicuramente sarà più avido di risorse.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	14	-	-
Expression	-	10	0	1558
FIFO	-	-	-	-
Instance	0	-	3196	3451
Memory	36	-	0	0
Multiplexer	-	-	-	321
Register	-	-	737	119
Total	36	24	3933	5449
Available	280	220	106400	53200
Utilization (%)	12	10	3	10

Figure 25: Risorse utilizzate dal modulo di rettificazione

Per quanto riguarda il codice eseguito su ARM è stata realizzata una libreria rectification.h che contiene le funzionalità minime per poter astrarre dai driver autogenerati da Vivado e per poter inviare i parametri ai moduli senza preoccuparsi del funzionamento a interrupt o dell'indirizzo in cui sono mappati i due moduli di rettificazione. In particolare è disponibile la funzione:

```
void rectification_configure(int master_slave,
int last_useful_row, int first_rectification_row,
short* table_x, short* table_y)
```

che consente di specificare il sensore da programmare (MASTER o SLAVE, oppure RIGHT o LEFT), i parametri di configurazione e gli indirizzi delle tabelle di rettificazione (separate in x e y). Sono disponibili infine altre due funzioni che sfruttano la rectification_configure: rectification_set() invia tabelle e parametri ai sensori (attualmente cablati nel codice nel file rectification_tables.h), rectification_unset() reimposta tutti i parametri a quelli di default, rendendo il modulo di rettificazione di fatto un filtro di

identità (utile per il debug).

Si riporta infine il risultato finale ottenuto rettificando le immagini dei sensori Aptina a runtime:

Figure 26: Immagini stereo prima di essere rettificate

Figure 27: Immagini stereo rettificate

7 Pre-processing: census ternaria

Tra i moduli di rettificazione ed SGM è stato posto un modulo di pre-filtraggio che implementa la trasformata Census. Tale funzione viene applicata ad una finestra di pixel, il cui centro coincide con il pixel in esame, e traduce tale finestra in una stringa di bit. La funzione di confronto utilizzata per ogni pixel è la stessa ed è estremamente semplice:

$$census(i) = 1 \quad if \quad I(p(i)) > I(p_c) \quad else \quad 0$$

dove I è l'intensità del pixel [0,255], p(i) l'i-esimo pixel della finestra (escluso quello centrale) e p_c il pixel centrale. Considerando una finestra quadrata, come tipicamente viene scelta, la trasformata Census per ogni pixel produce in uscita una stringa di NxN - 1 bit, dove N è la larghezza della finestra. La trasformata Census è quindi una trasformata locale e porta dei vantaggi (tolleranza al rumore e a trasformazioni monotone dell'intensità) in cambio di un costo computazionale molto basso: i confronti infatti possono essere eseguiti in parallelo e con poche risorse (calcoli tra interi), e il risultato in uscita resta comunque una stringa di bit di dimensione modesta.

La trasformata Census implementata in Vivado HLS utilizzata una finestra 5x5 e presenta due variazioni:

- La trasformata non viene calcolata su tutti i punti della finestra ma in un intervallo più ristretto (a scacchiera); in questo modo si perdono informazioni ma si ottine una stringa di bit più piccola (12 confronti contro 24) che potrà essere gestita in maniera più efficiente da SGM.

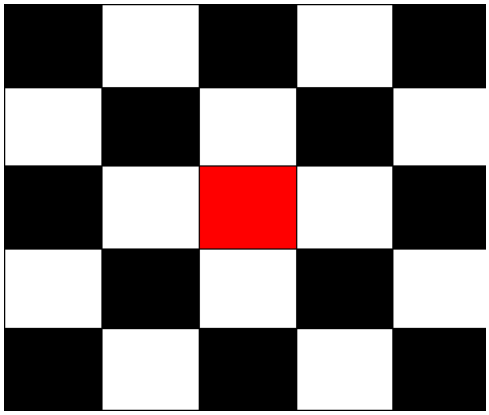


Figure 28: Finestra utilizzata per il calcolo della trasformata census (in nero i punti utilizzati)

- Viene utilizzata una variante dalla Census classica, chiamata Census ternaria, la cui funzione è:

$$census\ ternaria(i) = \begin{cases} a & if \quad -\epsilon \leq I(p(i) - I(p_c)| \leq \epsilon \\ b & if \quad I\big(p(i)\big) - I(p_c) > \epsilon \\ c & if \quad I\big(p(i)\big) - I(p_c) < -\epsilon \end{cases}$$

Figure 29: Definizione della census ternaria (a,b e c possono essere codificati a piacere)

Questa variante permette di introdurre un intervallo di incertezza (quello centrale, in quanto non c'è stata una variazione significativa di intensità) che potrà essere opportunamente gestito da SGM. Il parametro ϵ inoltre è stato reso configurabile via Axi-Lite e infine via ARM attraverso un'apposita funzione. Lo svantaggio è soltanto in termini di risorse, in quando ad ogni confronto sono associati 2 bit anzichè 1, quindi la stringa di uscita ritorna ai 24 bit originari.

A questo punto SGM non riceverà più in ingresso direttamente i pixel ma la stringa generata dalle due trasformate Census (left e right); la

funzione che calcolerà il costo locale dovrà quindi tenerlo in considerazione.

8 Implementazione di SGM

Nella camera già realizzata su scheda Spartan SGM era stato implementato per gestire un intervallo di disparità di 32 [0,31]; le scanline utilizzate erano solamente 4 (0,1,2,3) in quanto l'unico modo per calcolare la funzione costo sulle altre scanline è leggere le immagini rettificate al contrario (dall'ultimo pixel ricevuto al primo); per effettuare questo passaggio è necessario una memoria di appoggio su cui salvare l'intera immagine ma soprattutto i costi aggregati durante la prima lettura. Sulla scheda utilizzata ciò non era realizzabile, in quanto l'unica memoria disponibile era la BRAM (che è per sua natura di dimensione limitata), mentre con lo Zynq è possibile appoggiarsi alla memoria DDR. Inoltre nello Zynq sono presenti più risorse sulla FPGA ed è quindi possibile allargare l'intervallo di disparità per permettere alla camera di riconoscere correttamente la depth di oggetti più vicini. Il traguardo finale da raggiungere è infatti quello di implementare una pipeline con SGM a 128 disparità e capace di salvare in memoria DDR le immagini rettificate e i costi aggregati calcolati nella prima lettura, per poi rileggere le immagini (e i costi aggregati) al contrario, effettuare il calcolo sulle altre 4 scanline e ottenere una mappa di disparità molto più accurata.

Nella prima fase di lavoro ci si è focalizzati sul realizzare una versione di SGM ottimizzata quanto più possibile per l'implementazione in FPGA. In seguito l'attenzione si è spostata nel costruire tutto il resto dell'architettura necessaria ad effettuare la rilettura dei valori sopra descritti.

8.1 Struttura di SGM e algoritmo

Il modulo SGM da realizzare prende in ingresso i due axi-stream a 24 bit generati dai due moduli Census, effettua il matching lungo le scanline 0,1,2,3 e restituisce un axi-stream contenente i costi aggregati calcolati.

I passaggi da effettuare per ogni pixel sono sempre gli stessi:

1. Leggere i valori in ingresso ed effettuare lo shift del line buffer interno contenente i D valori precedenti relativi all'immagine right (dove D è la disparità).

2. Calcolare i costi puntuali, che non dipendono dalla particolare scanline

3. Recuperare i costi globali relativi alle scanline 1,2 e 3 dalla memoria BRAM

4. Per ogni scanline calcolare (in parallelo) i nuovi costi globali e il minimo di essi (in quanto servirà alla successiva iterazione)

5. Effettuare una fase di writeback nella BRAM dei costi relativi alle scanline 1,2 e 3

6. Aggregare i costi delle 4 scanline e scriverli in uscita

E' inoltre possibile semplificare il calcolo del fattore di smoothness di SGM; la formula prevede infatti di calcolare un minimo tra D elementi, e a D - 3 di quegli elementi viene sommato lo stesso peso P2. Ciò equivale di fatto a calcolare il minimo tra soli 4 valori, sostituendo l'ultimo termine con il minimo dei costi globali relativi all'iterazione precedente. Di fatto l'algoritmo viene semplificato ma il problema è stato soltanto aggirato, in quanto sarà comunque necessario calcolare il minimo tra D elementi.

Considerando che le operazioni di calcolo effettuate da SGM sono molte e alcune di esse sono abbastanza critiche, si è scelto di implementare SGM in modo da operare ad una frequenza di clock almeno 4 volte superiore a quella dei sensori (quindi ad almeno 100 MHz). In questo modo è possibile fare in modo di avere in uscita un valori ogni 4 clock.

Nelle prossime sezioni vengono riportare le considerazioni più significative riguardo l'implementazione specifica di SGM e alcune ottimizzazioni effettuate.

8.2 Calcolo dei costi locali

La funzione che si occupa di calcolare i costi locali deve comparare le stringhe di bit generate dalla trasformata Census e stabilire quando queste sono più o meno simili. Ciò porta immediatamente ad utilizzare la distanza di Hamming

come metro di somiglianza. Utilizzando la Census ternaria, e avendo quindi 3 possibili valori in uscita, è possibile scegliere quando considerare similii due stringhe di due bit. In base all'analisi delle immagini generate successivamente, si è scelto di definire simili due stringhe di due bit soltanto quando queste sono identiche tra di loro; in questo modo pixel che nella Census erano stati considerati di fasce diverse verranno qui considerati diversi.

L'utilizzo della distanza di Hamming porta alcuni vantaggi e svantaggi:

- Il valore massimo che è possibile ottenere dalla distanza di Hamming è 12 (cioè quando tutte le stringhe di due bit sono ritenute simili). Quindi è possibile codificare i costi locali con soltanto 4 bit.

- La distanza di Hamming per sua natura è un'operazione sequenziale (in quanto è necessario incrementare un contatore ogni volta che si verifica una somiglianza tra stringhe). Per sfruttare il parallelismo ed aumentare le prestazioni il calcolo è stato diviso in più moduli, ognuno dei quali opera all'interno di una sottoregione della stringa totale; i contatori di ogni modulo vengono infine sommati e costituiscono il costo locale.

8.3 Calcolo del minimo dei costi globali

Il problema più critico per quanto riguarda le risorse utilizzate e il rispetto dei timing è costituito dal calcolo del valore minimo tra i costi globali. Il problema nasce dal fatto che il calcolo del valore minimo di un array è un'operazione strettamente sequenziale, in quanto ad ogni iterazione è necessario conoscere il minimo dei valori precedenti. Codificando il calcolo del minimo nella maniera in cui si sarebbe scritto in normale codice C, il timing totale di SGM arriva fino a 404 ns, contro i 10 ns che sono stati imposti come target.

Per risolvere questo problema è stato necessario costruire una struttura ad albero binario le cui foglie sono i costi globali; in questo modo è possibile calcolare il minimo attraverso successive iterazioni in cui il minimo tra due nodi fratelli viene inserito nel padre. Risalenado ad ogni passo la gerarchia si troverà infine il minimo nel nodo radice. Rispetto alla struttura sequenziale originaria è possibile sfruttare al massimo il parallelismo: se i costi globali sono 128, con 64 comparatori è possibile calcolare in parallelo 64 minimi.

Nelle successive iterazioni sarà inoltre possibile utilizzare gli stessi comparatori per calcolare gli altri minimi.

8.4 Gestione della BRAM

Per accedere ai costi globali delle scanline 1, 2 e 3 è necessario memorizzare tali costi per una riga intera. La BRAM viene appunto utilizzata a tale scopo e contiene un'intera riga di costi globali, per un totale di 3 x D x WIDTH x GLOBAL_COST_BIT_WIDTH bit (con D la disparità). Per superare il vincolo delle porte limitate per l'accesso alla BRAM i costi globali correnti vengono salvati in un temporaneo shift-register sintetizzato con le LUT e i FF (che consente quindi un accesso rapido e in parallelo). Il vantaggio dello shift-register è che per ogni pixel è necessario caricare dalla BRAM soltanto i costi globali della scanline 3 (in quanto i costi globali della scanline 3 nel punto precedente diventano i costi globali della scanline 2 nel punto corrente, e così via). Una volta calcolati i nuovi costi globali per il pixel corrente sarà necessaria una fase di writeback in cui salvare tali costi in BRAM.

8.5 Risorse utilizzate

Si riportano le risorse stimate da Vivado HLS per la sintesi di SGM con disparità rispettivamente 32, 64 e 128. Si noti che la reale implementazione in Vivado ha sempre portato ad un calo di qualche punto percentuale delle risorse (in particolare delle LUT).

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	807
FIFO	-	-	-	-
Instance	0	-	1410	3027
Memory	22	-	0	0
Multiplexer	-	-	-	979
Register	-	-	2981	11
Total	22	0	4391	4824
Available	280	220	106400	53200
Utilization (%)	7	0	4	9

Figure 30: Risorse utilizzate da SGM con disparità 32

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	1506
FIFO	-	-	-	-
Instance	0	-	4020	6726
Memory	44	-	0	0
Multiplexer	-	-	-	883
Register	-	-	5382	11
Total	44	0	9402	9126
Available	280	220	106400	53200
Utilization (%)	15	0	8	17

Figure 31: Risorse utilizzate da SGM con disparità 64

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	2914
FIFO	-	-	-	-
Instance	0	-	8116	13350
Memory	86	-	0	0
Multiplexer	-	-	-	2423
Register	-	-	14912	785
Total	86	0	23028	19472
Available	280	220	106400	53200
Utilization (%)	30	0	21	36

Figure 32: Risorse utilizzate da SGM con disparità 128

8.6 Pipeline e risultati ottenuti

La pipeline finale realizzata in Vivado è composta da due parti:

1. La prima parte (pre-SGM) è identica per le camere left e right (ed è stata quindi duplicata) e comprende la rettificazione e il calcolo della trasformata census; è stato inoltre inserito un filtro di smooth (è un filtro di convoluzione che opera una media pesata dei pixel della finestra) che permette di ridurre il rumore in ingresso alla census.

2. La seconda parte è composto da SGM e da un modulo che, presi in ingresso i costi aggregati da SGM, produce in uscita la disparità. Quest'ultimo modulo si occupa essenzialmente di calcolare il minimo dei costi aggregati e in particolare il suo indice (che rappresenta appunto la disparità). Una volta calcolata la disparità, il flusso dei dati viene mandato in ingresso ad un axis_to_ddr_writer che si occupa di scrivere la mappa di disparità in memoria DDR. Essendo la disparità codificata con 8 bit, la mappa generata potrà essere trattata come una normale immagine senza modificare nulla.

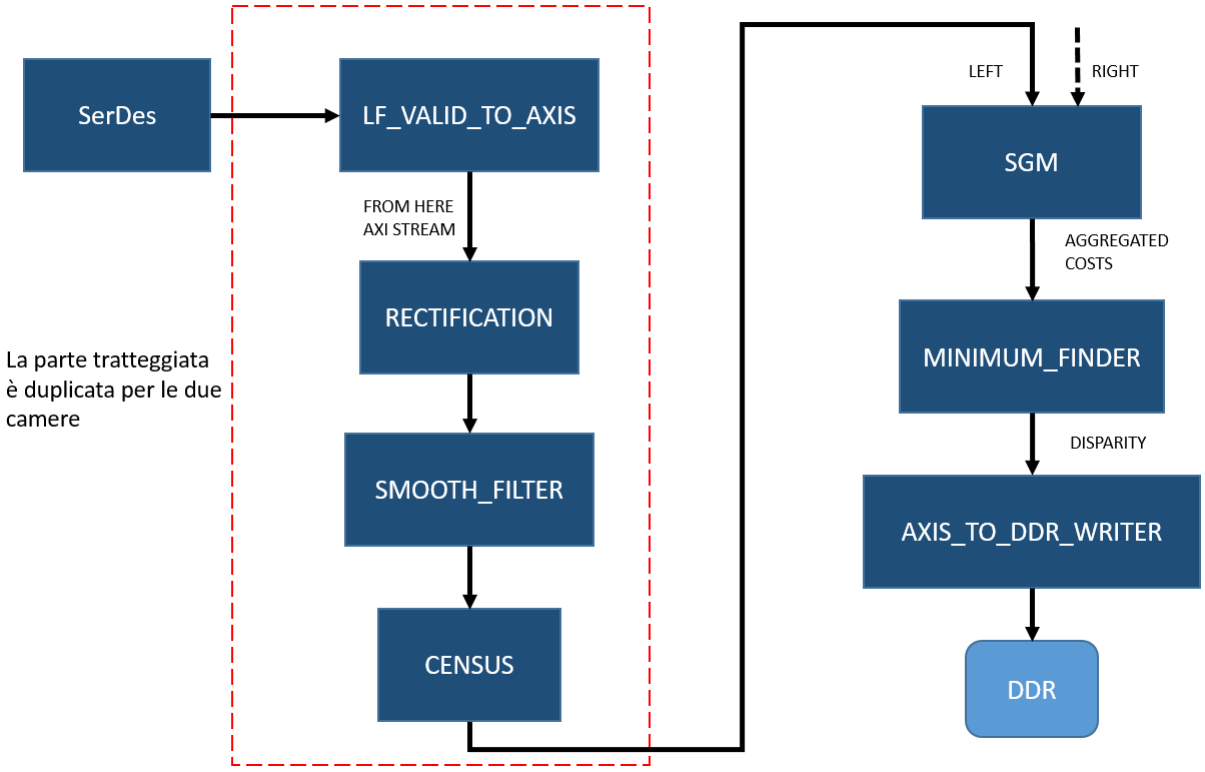


Figure 33: Pipeline di elaborazione completa con SGM

Si riportano le principali risorse utilizzate per l'intero progetto implementato (SGM con disparità 128):

Name	1 Slice LUTs (53200)	Slice Registers (106400)	Block RAM Tile (140)
design_1_wrapper	57.26%	41.12%	66.43%
design_1_i (design_1)	57.26%	41.12%	66.43%
SGM (design_1_SGM_0_1)	25.44%	21.97%	30.71%
rectification_master (design_1_rectif...	5.74%	2.20%	12.86%
rectification_slave (design_1_rectific...	5.73%	2.20%	12.86%
axi_mem_intercon_reader_disparity ...	4.05%	3.23%	0.00%
writer_rectification (writer_rectificati...	3.79%	2.96%	2.14%
processing_system7_axi_periph (de...	3.67%	3.28%	0.00%
MinimumFinder (design_1_MinimumFi...	2.49%	0.77%	0.00%
axis_to_ddr_writer_disparity (design...	1.26%	1.09%	0.71%
ddr_to_axis_reader_disparity (desig...	1.11%	0.96%	0.71%
census_transform_master (design_1...	0.66%	0.28%	1.43%
census_transform_slave (design_1_...	0.66%	0.28%	1.43%
serdes_v5 (design_1_serdes_v5_0_0)	0.48%	0.25%	0.00%
axi_mem_intercon_writer_disparity (...	0.37%	0.27%	0.00%
FiltroSmooth_Master (design_1_Filtr...	0.29%	0.17%	0.71%
FiltroSmooth_Slave (design_1_Filtro...	0.28%	0.17%	0.71%

Figure 34: Risorse utilizzate in FPGA con SGM a disparità 128

Le prossime immagini mostrano i risultati ottenuti implementando SGM rispettivamente con disparità 32, 64 e 128. Qui era ancora utilizzata la trasformata census calssica (binaria). Come si può vedere l'aumento di disparità consente sia di rilevare correttamente oggetti più vicini ma anche di ridurre notevolmente il rumore.

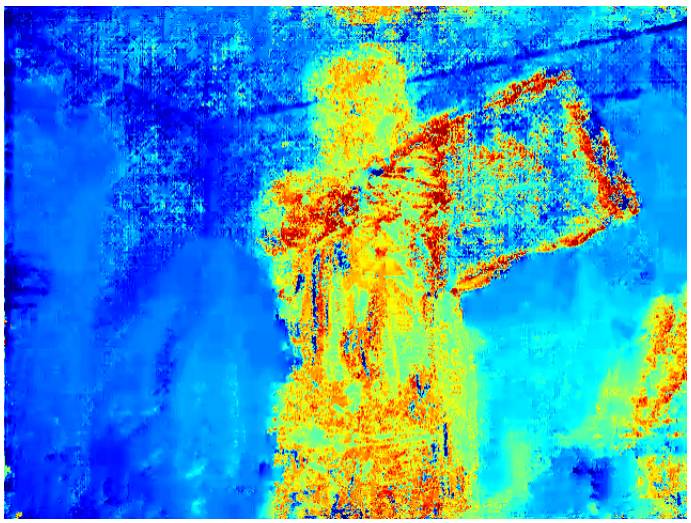


Figure 35: Mappa di disparità con SGM a disparità 32

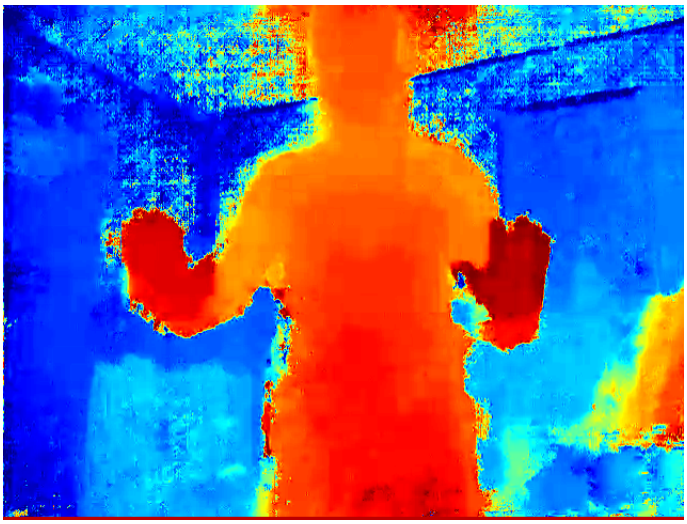


Figure 36: Mappa di disparità con SGM a disparità 64

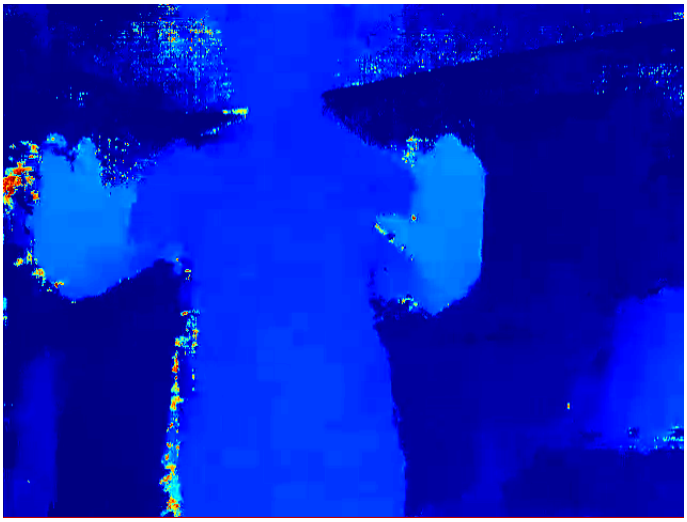


Figure 37: Mappa di disparità con SGM a disparità 128

Nell'immagine con disparità 32 si può notare come già a qualche metro di distanza l'algoritmo non produca risultati soddisfacenti. Con disparità 128 è invece possibile avvicinarsi a poche decine di centimetri dai sensori, come nell'immagine seguente:

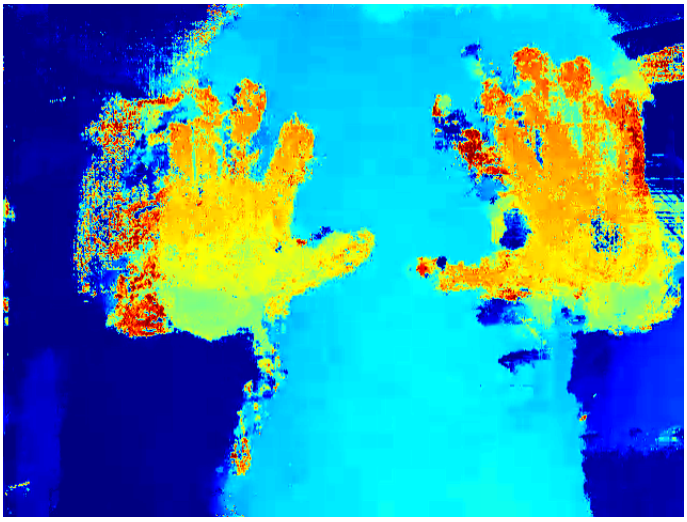


Figure 38: Mappa di disparità con SGM a disparità 128 in prossimità della stereo camera

Nelle prossime immagini è stata invece introdotta la census ternaria. I valori di epsilon (dimensione della regione di incertezza) sono rispettivamente 1, 3, 5 e 15. Come si può notare il passaggio alla census ternaria produce mappe di migliore qualità; al crescere della regione di incertezza però diminuisce la sensibilità di SGM nel rilevare le variazioni di profondità, generando così immagini più uniformi e meno corrette. E' stato quindi scelto di mantenere un valore di epsilon basso (tipicamente 1).

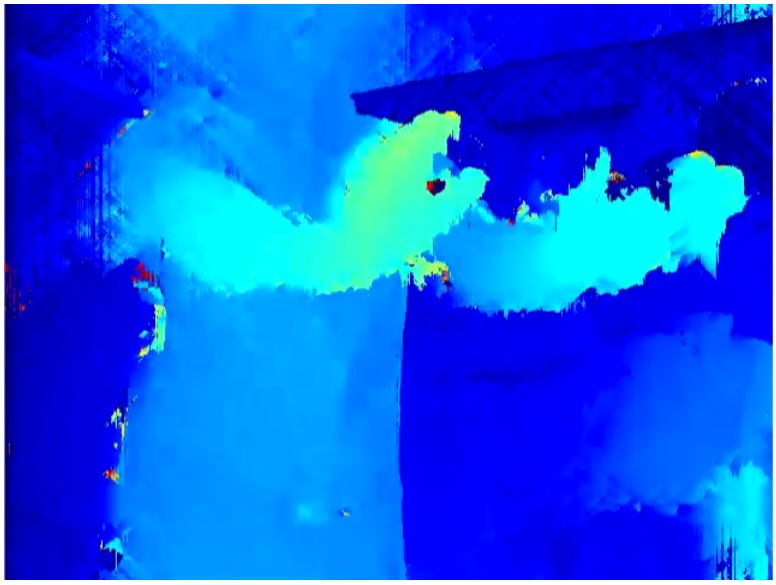


Figure 39: Mappa di disparità con census ternaria e epsilon = 1

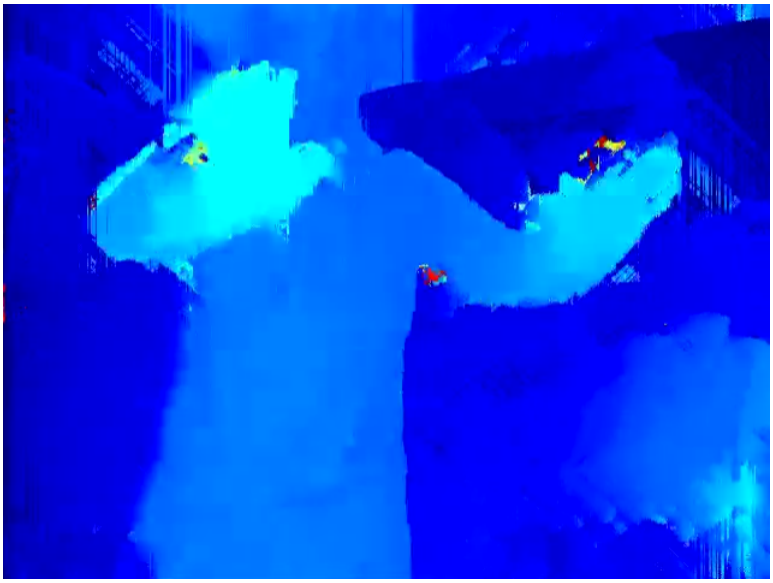


Figure 40: Mappa di disparità con census ternaria e epsilon = 3

Figure 41: Mappa di disparità con census ternaria e epsilon = 5

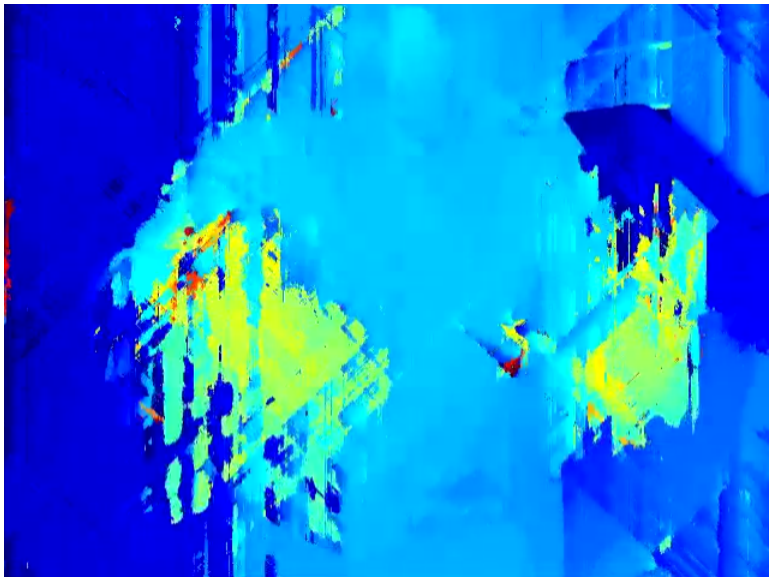


Figure 42: Mappa di disparità con census ternaria e epsilon = 15

9 SGM a 8 scanline

Rispetto alla pipeline già realizzata è necessario aggiungere due stadi importanti per poter utilizzare tutte le 8 scanline di SGM:

1. Scrivere le immagini rettificate in memoria ("costano" meno che scrivere i dati in uscita dalla trasformata census) per poi rileggerle al contrario e reiniettarle in SGM

2. Salvare i costi aggregati in uscita dalla prima fase di SGM in DDR per poi rileggerli (anch'essi al contrario) e sommarli ai costi calcolati durante la seconda fase. La scrittura e lettura dei costi aggregati rappresenta il problema più critico da superare: considerando infatti un frame rate di 30 immagini al secondo, una width di 8 bit per i costi aggregati e disparità 128, è necessario una banda di $30 \times 640 \times 480 \times 128 \times 8bit/s = 9.4Gb/s$ per la scrittura e la lettura di tali costi.

Oltre a questi stadi sono stati inseriti altri moduli di raccordo che verranno descritti in seguito.

9.1 Parte Pre-SGM: Inversione delle immagini rettificate

La pipeline precedente a SGM è ancora una volta identica per entrambe le camere e sarà pertanto duplicata.

Si può subito notare che, per costruzione, non sarà possibile calcolare la mappa di disparità per ogni coppia di immagini generate dai sensori in quanto tale mappa sarà generata attraverso due iterazioni consecutive sulle stesse immagini. Pertanto è stato inserito un modulo a monte di tutta la pipeline, chiamato *1ImmagineSu2*, il cui scopo è appunto filtrare le immagini in ingresso campionandone una ogni due in maniera alternata. In questo modo tutta la pipeline opererà soltanto sulle immagini significative per il calcolo della mappa di disparità senza effettuare alcuna modifica.

Per realizzare l'inversione dell'immagine rettificata è stato realizzato il modulo *ImageInverter* (la cui struttura è basata sui già realizzati axis_to_ddr_writer e ddr_to_axis_reader), il quale scrive l'immagine rettificata in DDR e la rilegge al contrario (ovvero partendo dall'ultimo indirizzo dove aveva scritto

gli ultimi pixel dell'immagine). Attualmente il modulo scrive e legge sempre all'interno della stessa area di memoria, cioè non utilizza un frame buffer.

Sono stati infine inseriti due moduli per completare lo scheletro della pipeline:

- Il modulo *AxiStreamMux* riceve due axi stream in ingresso, relativi all'immagine rettificata invertita a non. Alla prima iterazione fa passare l'immagine non invertita, mentre alla seconda iterazione manda in uscita l'immagine invertita.
- Il moduli *Axi Broadcaster* di Xilinx riceve l'immagine rettificata in input e la manda in uscita su due axi stream differenti, il primo collegato al mux e il secondo collegato all'ImageInverter.

Di seguito viene schematizzato il funzionamento della pipeline pre-SGM durante la prima fase e durante la rilettura della immagini (in rosso sono indicati gli stream attivi, in nero quelli non attivi).

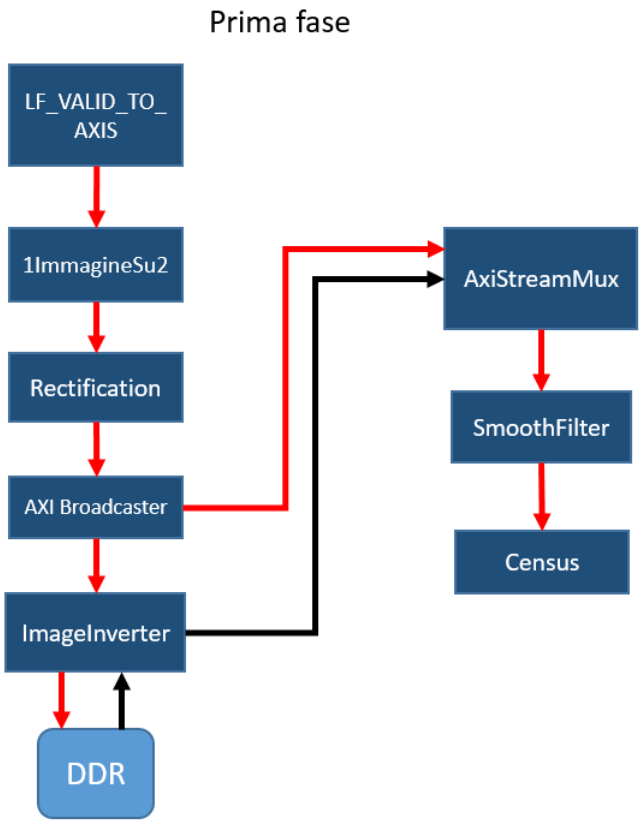


Figure 43: Pipeline Pre-SGM durante la prima fase (rettificazione e salvataggio in DDR)

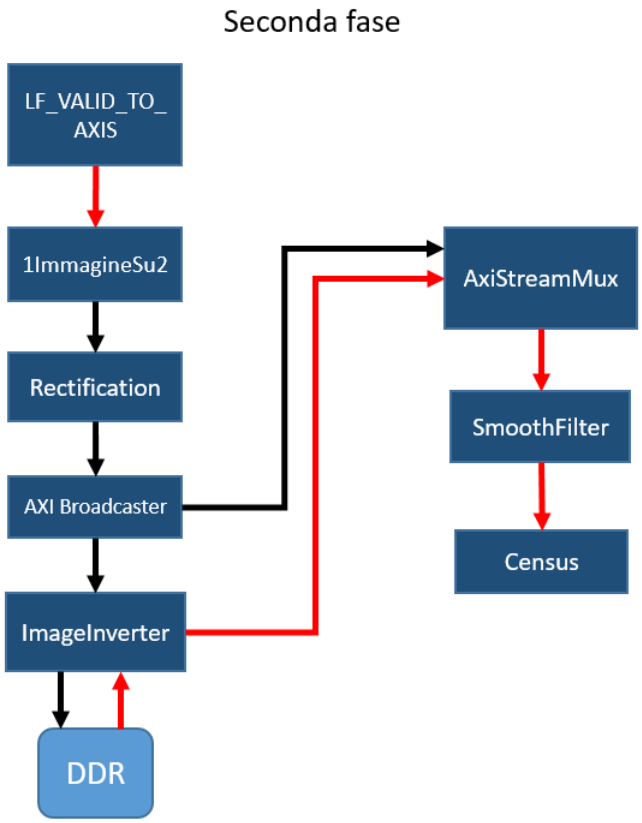


Figure 44: Pipeline Pre-SGM durante la seconda fase (rilettura al contrario delle immagini)

9.2 Modifiche a SGM

La costruzione della pipeline finora realizzata non è ancora sufficiente a garantire il corretto funzionamento di SGM durante la seconda fase. Si supponga infatti che SGM abbia calcolato i costi aggregati dell'ultimo pixel durante la prima fase; in questo momento nel line buffer interno a SGM sono presenti i punti compresi nell'intervallo [WxH - D, WxH - 1]. E' possibile fare due osservazioni riguardanti il momento in cui arriverà l'immagine invertita:

1. Il line buffer non dovrà più spostarsi verso destra, come accade normalmente, ma verso sinistra.

2. All'arrivo del primo pixel dell'immagine left tutti i pixel dell'immagine right per calcolare i costi saranno già presenti. All'arrivo del secondo pixel non sarà necessario campionare il secondo pixel dell'immagine right (così come accade nella prima fase), bensì il pixel in posizione WxH - D - 1. Bisogna quindi il qualche modo anticipare l'immagine invertita right rispetto a quella left.

Le figure 45 e 46 schematizzano il funzionamento del line buffer di SGM durante le due fasi:

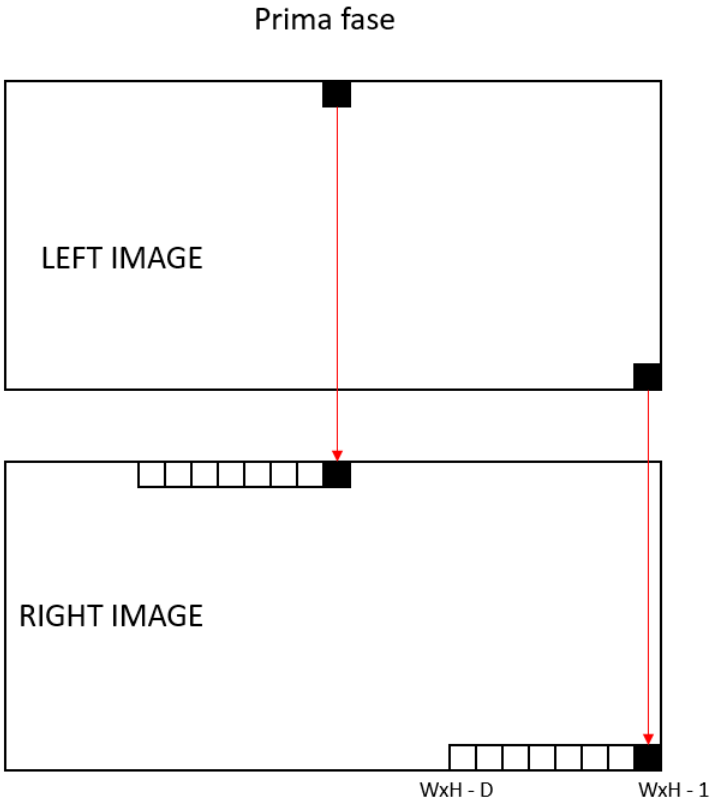


Figure 45: Line buffer di SGM durante la prima fase (in nero i pixel campionati contemporaneamente)

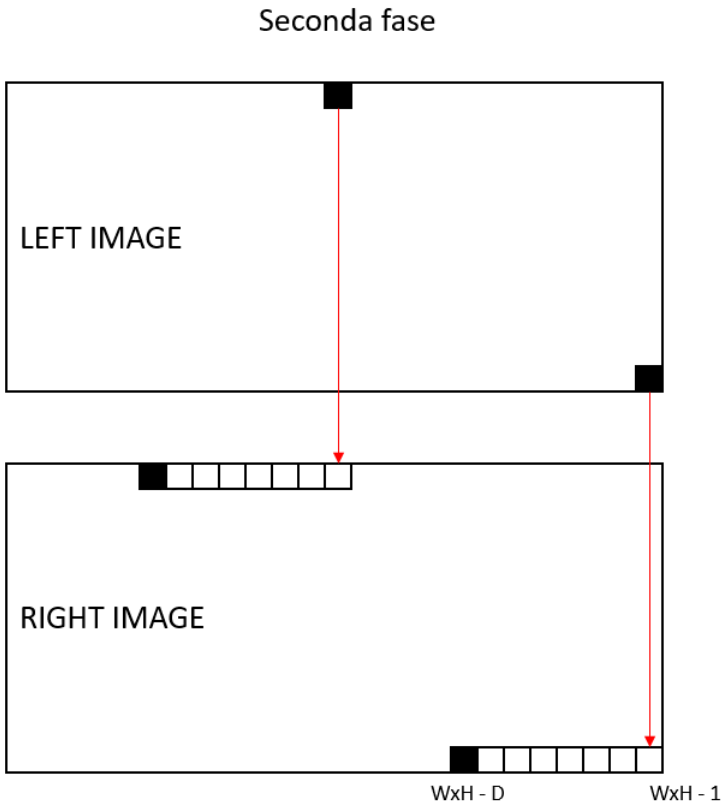


Figure 46: Line buffer di SGM durante la seconda fase (in nero i pixel campionati contemporaneamente)

Il primo problema è stato risolto imponendo a SGM di effettuare durante la rilettura della immagini uno shift a sinistra del line buffer; questa è l'unica modifica effettuata a SGM. Per garantire l'anticipo dell'immagine right rispetto alla left è stato introdotto il modulo *RightImageSynchronized* nella pipeline precedente a SGM e soltanto nella parte relativa all'immagine right (tra i moduli ImageInverter e AxiStreamMux). L'IP realizzato scarta i primi D - 1 valori dell'immagine right invertita in modo che il primo pixel in uscita sia proprio WxH - D (che è già presente in SGM ma che è comunque necessario campionare assiame al valore left). Si noti che questo è l'unico modulo che spezza la simmetria tra pipeline left e right prima di SGM; inoltre è dipendente dal valore di disparità che si vuole implementare.

La pipeline fin qui realizzata è quindi in grado di generare le mappe di disparità anche per l'immagine invertita. Inserendo i moduli già realizzati a valle di SGM è possibile visualizzare entrambe le mappe di disparità in maniera alternata.

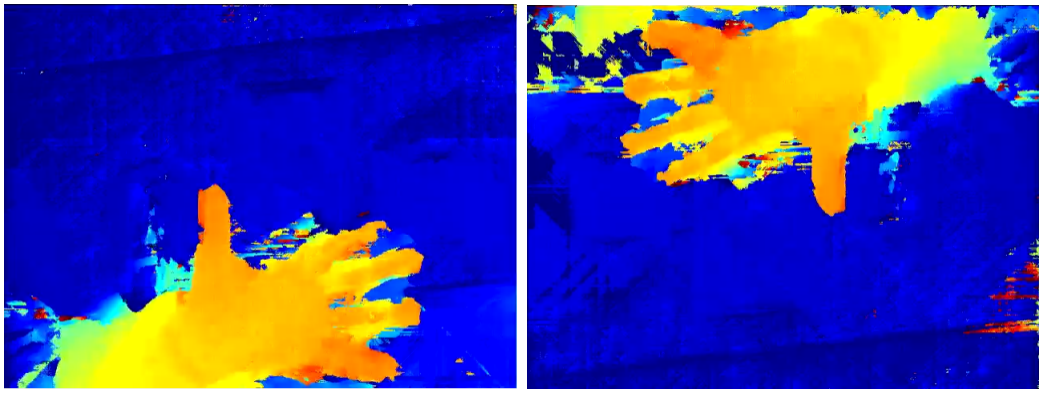


Figure 47: Mappe di disparità generate successivamente da SGM sulla stessa coppia di immagini stereo. La prima fa riferimento alle scanline 0,1,2,3 e la seconda alle scanline 4,5,6,7, ottenute grazie all'inversione delle immagini di partenza.

9.3 Parte Post-SGM: rilettura dei costi aggregati e somma delle 8 scanline

La pipeline a valle di SGM ha essenzialmente due compiti:

1. Durante la prima fase deve scrivere i costi aggregati in uscita da SGM nella memoria DDR.

2. Durante la seconda fase deve rileggere tali costi (dall'ultimo al primo), sommarli ai costi calcolati contemporaneamente da SGM sull'immagine invertita, calcolarne il minimo (ovvero la disparità) e salvare il risultato in DDR.

Innanzitutto sono stati realizzati i moduli che compongono lo scheletro della pipeline, disinteressandosi del trasferimento dei costi aggregati in memoria; in particolare sono stati realizzati i seguenti moduli:

- Il modulo *DsiAdder* somma i costi aggregati durante la seconda fase e li invia al MinimumFinder già realizzato.

- Il modulo *DsiDemux* si occupa di inviare i costi aggregati ai giusti componenti: durante la prima fase vengono inviati alla parte di trasferimento dei costi aggregati in memoria, durante la seconda fase vengono inviati direttamente al DsiAdder.

Viene qui schematizzato il funzionamento della pipeline post-SGM rispettivamente durante la prima e la seconda fase (in rosso gli stream attivi, in nero gli stream non utilizzati):

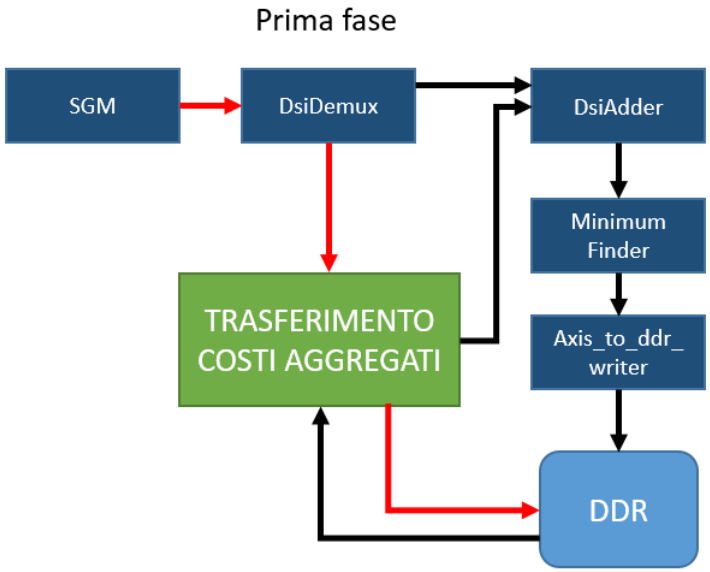


Figure 48: Pipeline Post-SGM durante la prima fase (salvataggio dei costi aggregati)

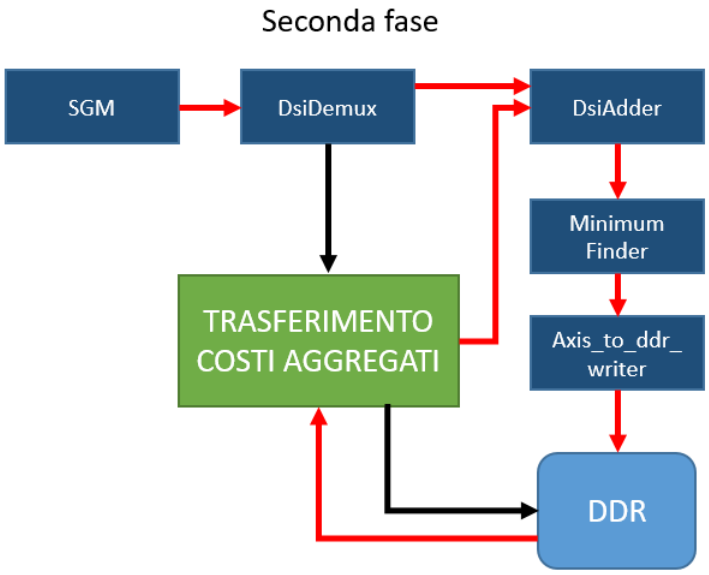


Figure 49: Pipeline Post-SGM durante la seconda fase (rilettura dei costi aggregati, somma e calcolo della disparità)

Per quanto riguarda il trasferimento vero è proprio, non è possibile utilizzare un normale axis_to_ddr_writer. Si ricorda infatti che con disparità 128 e un frame rate di 30 immagini al secondo è necessario avere una banda di 9.4 Gb/s per la scrittura e la lettura dei costi aggregati. Per raggiungere una banda tanto elevata è necessario utilizzare tutte e 4 le porte HP e 4 moduli, ognuno collegato ad una porta HP, per sfruttare il parallelismo.

Il modulo che si occupa della scrittura e rilettura al contrario dei costi aggregati è stato chiamato *fast_ dsi_ writer*; la struttura è simile all'ImageInverter: ogni fast_dsi_writer riceve in ingresso una porzione dei costi aggregati e li salva in DDR. Finita questa prima parte i costi vengono riletti e scritti in uscita. Come gli altri moduli di scrittura/lettura in DDR questo modulo contiene un buffer interno che viene copiato in memoria con trasferimento burst nel momento in cui viene riempito; nel fast_dsi_writer il dimensionamento di questo buffer risulta particolarmente importante, in quanto deve essere abbastanza grande da consentire ad un modulo di scrivere i costi aggregati in memoria nel tempo in cui gli altri tre moduli riempono i loro buffer interni. In

questo modo è possibile evitare stalli e sfruttare a pieno la banda offerta dalle porte HP. Allo stesso tempo è necessario considerare che i valori da salvare nel buffer sono molto grandi (1024 bit contro gli 8 bit di una normale immagine), quindi non sarà possibile utilizzare dei buffer eccessivamente grandi.

Come già enunciato, per scrivere nella maniera più efficiente possibile i costi aggregati è necessario riempire ciclicamente i 4 buffer dei 4 fast_dsi_writer; in questo modo quando un buffer è pieno il fast_dsi_writer e la pipeline possono continuare a lavorare in parallelo: il primo scriverà il contenuto del buffer in memoria, la seconda riempirà i buffer degli altri 3 moduli. Per implementare questo comportamento sono stati realizzati due moduli, chiamati *Dsi_Router* e *Dsi_Router_Inverse*. Il loro comportamento è duale: il primo è posto prima dei 4 fast_dsi_writer e si occupa di inviare ciclicamente ad essi i costi aggregati (partendo dal fast_dsi_writer 0). Il secondo modulo è posto dopo i fast_dsi_writer e si occupa di leggere i costi riletti dai fast_dsi_writer in maniera ciclica, ma al contrario (ovvero partendo dall'ultimo fast_dsi_writer). Ovviamente per lavorare correttamente i due moduli devono conoscere la dimensione del buffer interno dei fast_dsi_writer.

Con l'aggiunta di questi moduli la scrittura e rilettura dei costi aggregati potrebbe dirsi completa. In realtà tale struttura non è implementabile per due motivi riguardanti le risorse:

1. I buffer interni dei fast_dsi_writer devono garantire un parallelismo molto alto (fino a 1024 bit nel caso di disparità 128). Ciò implica di usare un numero spropositato di BRAM: le BRAM consentono un parallelismo fino a 36 bit quindi per salvare anche un solo costo aggregato sono necessarie 1024 / 36 = 29 blocchi. Considerando che sono istanziati 4 moduli, le BRAM utilizzati diventerebbero 116 su 140, un valore certamente non accettabile.

2. Il trasferimento in DDR di valori con parallelismo molto alto (1024 bit ma anche 256 bit) porta alla creazione di moduli AXI Interconnect che utilizzano un numero molto alto di risorse (fino al 20/30%), rendendo la sintesi irrealizzabile.

Per risolvere questo problema si è scelto di ridurre il parallelismo attraverso l'uso dei moduli AXI Width Converter (proprietari Xilinx), che con-

sentono appunto di aumentare o ridurre il parallelismo dei dati. Ovviamente questa operazione implica l'utilizzo di più cicli di clock, quindi si è scelto di ridurre il parallelismo di un fattore 4 in quanto SGM genera un valore in uscita ogni 4 clock. In questo caso l'utilizzo di BRAM complessivo dei 4 fast_dsi_writer passa al 20% in quanto non è più necessario memorizzare valori a 1024 bit, ma a 256 bit. Infine si è scelto di mantenere i trasferimenti da/verso la DDR sempre a 64 bit in modo da ridurre le risorse necessarie a realizzare tale trasferimento.

Consideriamo per esempio i 1024 bit di costi aggregati in uscita da SGM con disparità 128: tali bit vengono inizialmente spezzati 4 stringhe di 256 bit e memorizzati in BRAM, poi vengono ulteriormente suddivisi durante il trasferimento verso la DDR, infine vengono riletti a 64 bit, salvati in BRAM ancora a 256 bit e poi ricostruiti a 1024 bit dall'AXI Width Converter.

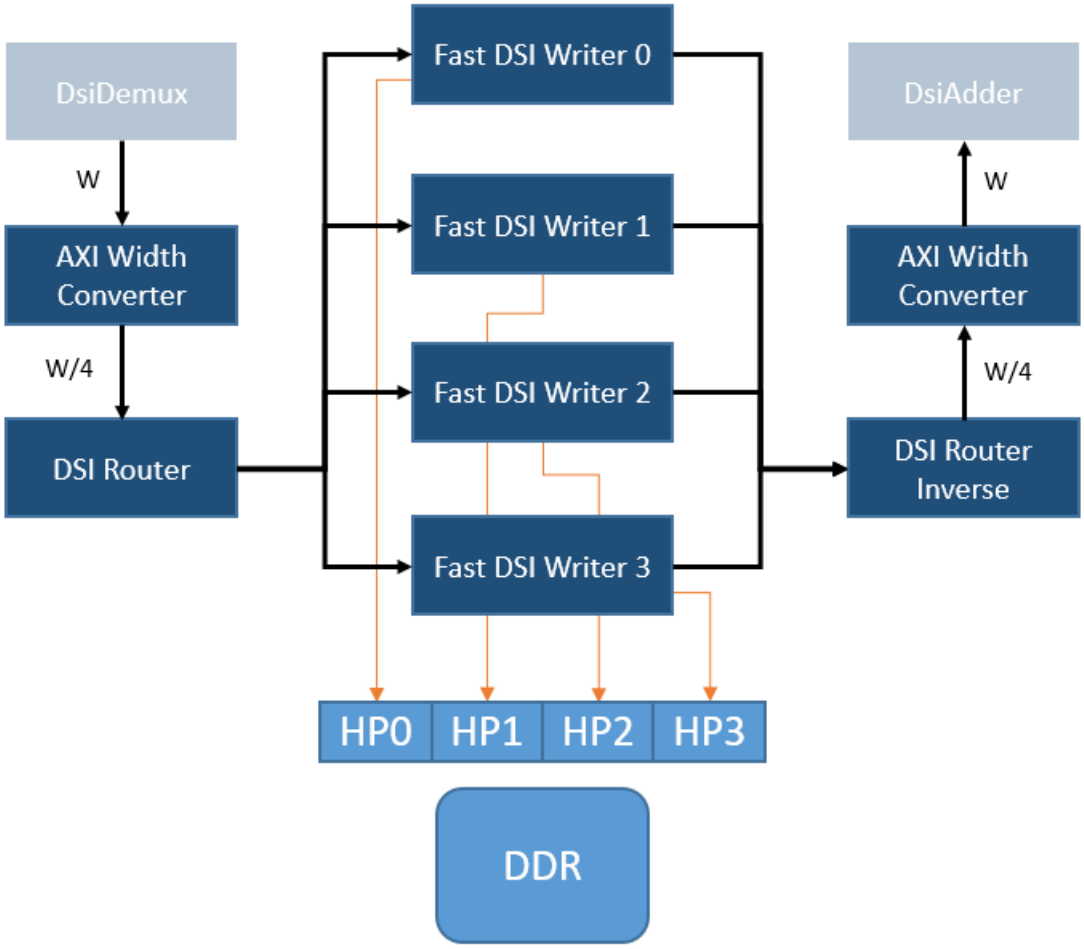


Figure 50: Pipeline per la scrittura e rilettura dei costi aggregati in parallelo sfruttando le 4 porte HP

Infine per la fase di debug sono stati inseriti due moduli *DummyDSI* che, se abilitati, forzano i costi aggregati a zero in uno dei due stream in ingresso al DsiAdder. In questo modo è possibile visualizzare sia il risultato totale (somma delle 8 scanline) oppure solo uno dei due risultati parziali, quello risultante dal matching diretto sull'immagine o quello risultante dal matching sull'immagine invertita.

9.4 Risultati ottenuti e sviluppi futuri

La pipeline completa è stata per adesso implementata con disparità 32. Questo valore ha permesso di concentrarsi sulla struttura della pipeline stessa piuttosto che sull'ottimizzazione delle risorse e dei timing.

Il progetto completo utilizza complessivamente 3 clock differenti:

1. La pipeline pre-SGM (rettificazione, census) opera alla stessa frequenza di clock dei sensori, attualmente a 18 MHz
2. SGM utilizza un clock a 100 MHz
3. Tutti i moduli di trasferimento delle immagini o dei costi aggregati in memoria operano ad una frequenza di 150 MHz (nello studio svolto da Lorenzo Ciotti è risultata la frequenza di clock che garantisce la banda maggiore)

Le risorse occupate dall'intera pipeline rappresentano circa la metà delle risorse disponibili sulla scheda. Probabilmente il passaggio da disparità 32 a 128 porterà ad un aumento del 30/40% sia delle LUT che delle BRAM utilizzate.

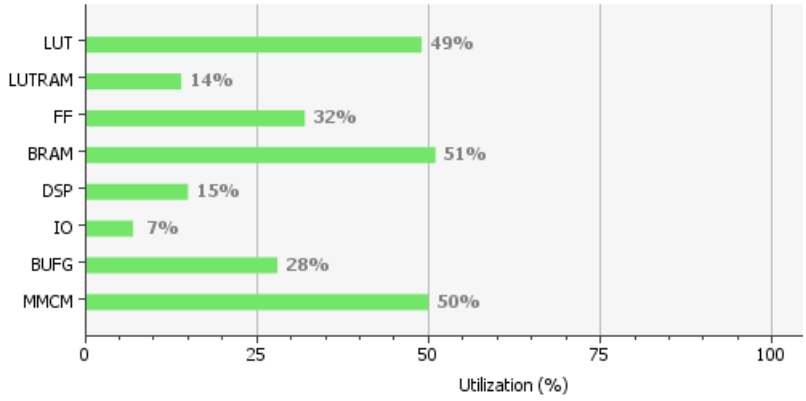


Figure 51: Risorse utilizzate dal progetto completo con disparità 32

Per quanto riguarda l'implementazione in FPGA, vi sono numerosi tempi di setup non rispettati. La causa probabilmente non è tanto da ricercare nel numero di risorse utilizzate (circa metà della BRAM e delle LUT), quanto nel fatto che molti moduli, in particolare i fast_dsi_writer e gli AXI Interconnect, necessitano una interazione stretta con la parte PS e in particolare

con la DDR e hanno dunque bisogno di essere sintetizzati in una regione il più possibile vicina al Processing System. Ciò crea probabilmente una congestione che porta Vivado a sintetizzare alcuni moduli troppo lontano e a non rispettare quindi alcuni timing. E' inoltre probabile che il non rispetto dei tempi di setup influenzi la qualità delle mappe generate (in particolare la consitenza dei costi aggregati).

Probabilmente con un attento studio sui timing, sulle direttive di Vivado e sulle strategy utilizzate dal tool per effettuare sintesi e implementazione sarà possibile superare questo ostacolo. Risolvere questo problema sarà fondamentale per consentire l'aumento di disparità, in quanto l'FPGA, con un 80% di risorse utilizzate, comincerà ad essere congestionata, rendendo l'implementazione certamente più problematica.

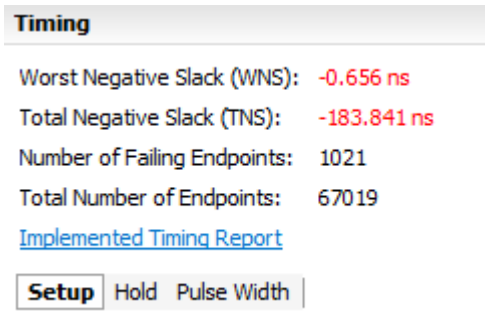
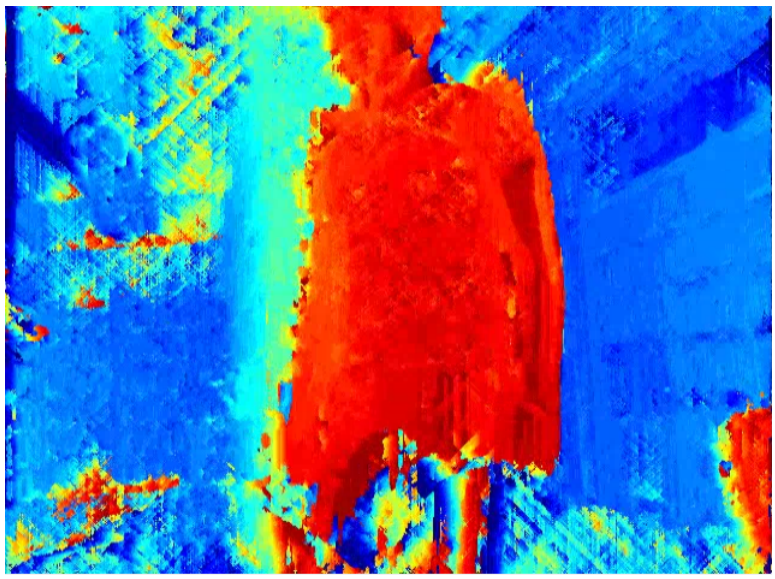
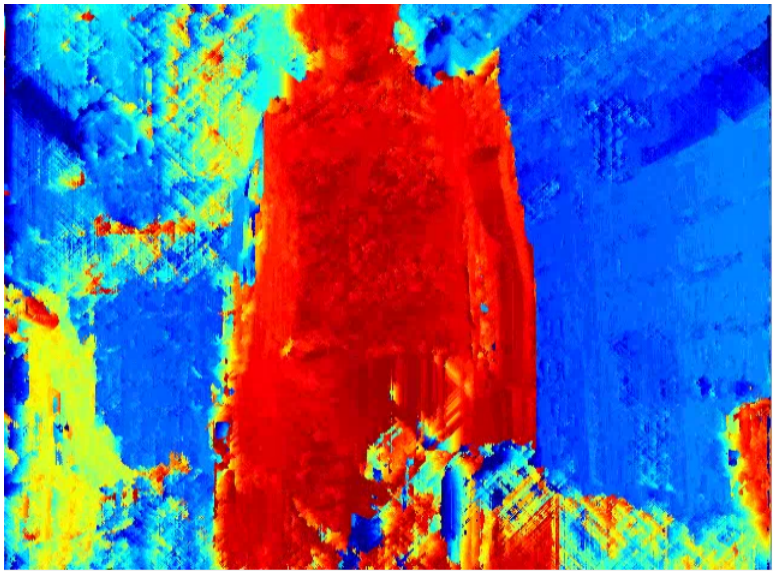


Figure 52: Tempi di setup non rispettati in fase di implementazione

I risultati infine ottenuti utilizzando tutte e 8 le scanline sono indice di una qualità decisamente migliore delle mappe di disparità ottenute. Come si può vedere dalle immagini riportate, l'algoritmo è in grado di riconoscere con maggiore accuratezza le variazioni di profondità e i bordi degli oggetti.



(a)



(b)

Figure 53: Mappa di disparità ricavata utilizzando solo le scanline 0,1,2,3 (a) o solo le scanline 4,5,6,7 (b)

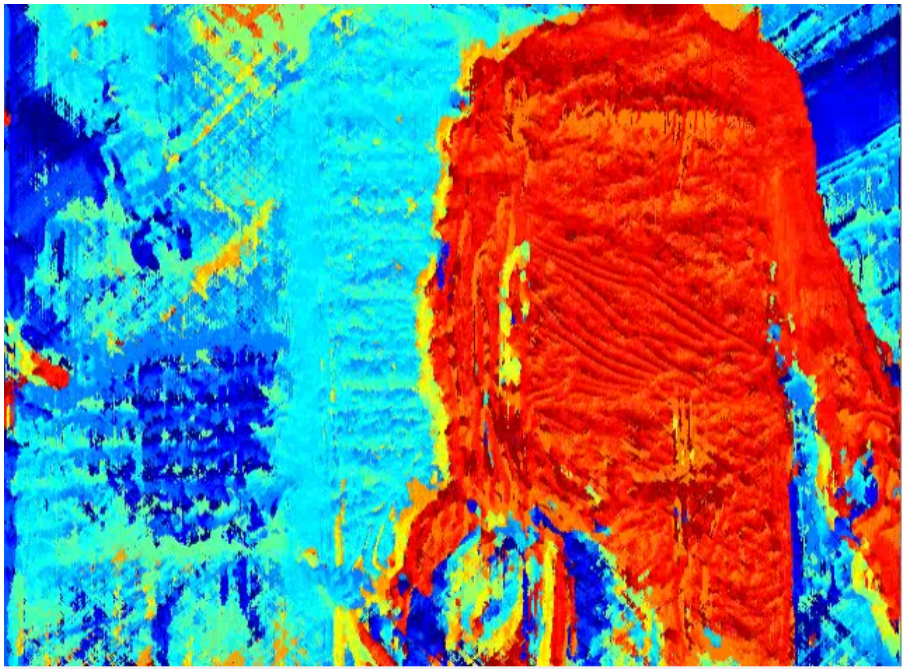


Figure 54: Mappa di disparità ricavata con l'utilizzo di tutte e 8 le scanline: viene riconosciuto con maggiore precisione l'ostacolo centrale (uno scaffale); inoltre vengono addirittura rilevate le pieghe dei vestiti della persona

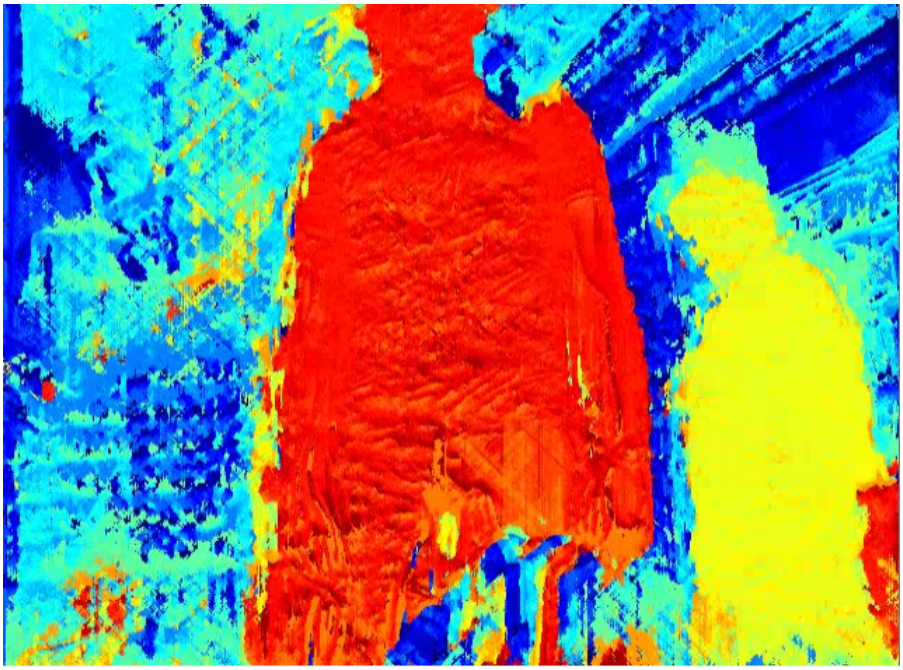


Figure 55: Mappa di disparità ricavata con l'utilizzo di tutte e 8 le scanline: i bordi sono individuati in maniera molto netta (si osservi lo spazio tra le due persone che non viene modificato dalla zona circostante a disparità elevata)

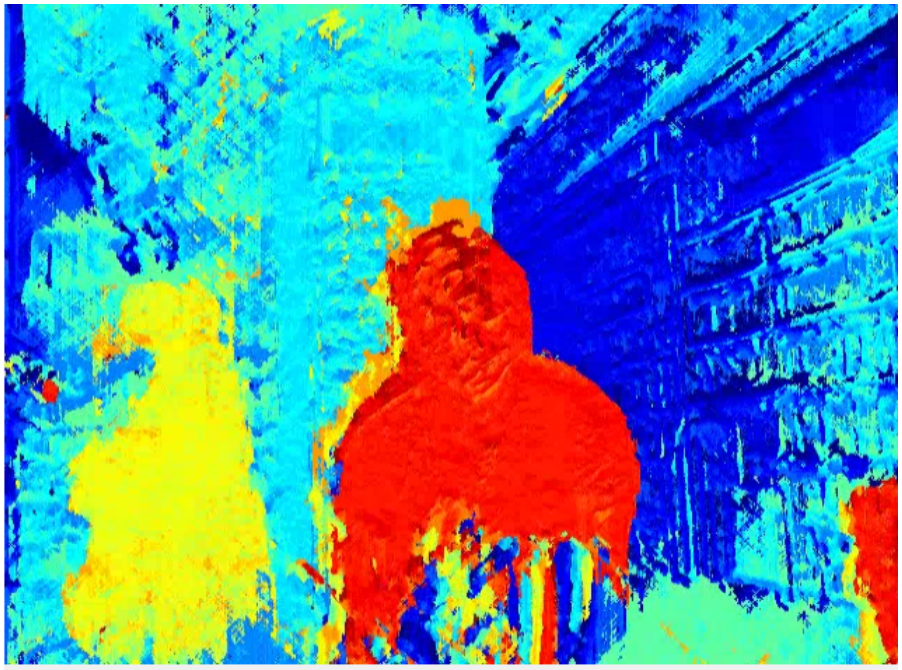


Figure 56: Mappa di disparità ricavata con l'utilizzo di tutte e 8 le scanline: viene riconosciuto molto accuratamente il volto e vengono rilevati i libri nello scaffale nella parte destra dell'immagine; con sole 4 scanline veniva percepita come una superficie quasi uniforme

Part V
CONCLUSIONI

Il lavoro di tesi ha permesso di costruire una pipeline completa di visione stereo; per realizzarla ho duvto affrontare per la prima volta il mondo dei sistemi embedded e in particolare quello delle FPGA e della programmazione hardware. Personalmente ritengo che entrambi siano ambienti affascinanti e consentano al programmatore di mettere in gioco tutta la sua creatività, sia per quanto riguarda la realizzazione di sistemi articolati, sia per quanto riguarda la ricerca della massima ottimizzazione di algoritmi complessi (come la rettificazione o SGM), che spesso porta a ripensare in maniera differente alcuni aspetti o comportamenti del sistema da realizzare.

Per quanto riguarda i risultati ottenuti, sono stati realizzati due progetti su FPGA basati su stereo matching: il primo usa SGM con disparità 128 e soltanto 4 scanline, il secondo usa SGM con disparità 32 e tutte e 8 le scanline. Il primo progetto può dirsi completo e permette di implementare altri algoritmi in hardware per accelerare altre funzionalità desiderate; il secondo progetto, una volta raggiunta la disparità 128, probabilmente non consentirà di aggiungere altri moduli in FPGA, ma la qualità delle mappe di disparità generate sarà notevolmente migliore.

Gli sviluppi immediatamente futuri sul progetto in FPGA sono i seguenti:

- Portare il progetto con SGM a 8 scanline da disparità 32 a 128. I moduli sono già pronti nelle versioni a disparità 32, 64 e 128, ma l'ostacolo maggiore sarà gestire i problemi di timing e la congestione della FPGA che hanno già cominciato a manifestarsi nella realizzazione a disparità 32.

- Realizzare la pipeline di post-processing (sia nel primo che nel secondo progetto), che consente di ottenere mappe di disparità più accurate.

Come già sottolineato, l'aumento di disparità consente alla camera di rilevare correttamente oggetti più vicini; sono in corso progetti di ricerca per la realizzazione di un sistema di guida autonomo che consenta di stimare la crescita e la maturazione della frutta all'interno di un'impresa agricola. Per poter effettuare stime su oggetti così piccoli è necessario avvicinarsi molto

e l'aumento della disparità diventa fondamentale per poter ricavare informazioni utili alle analisi da svolgere.

Altre attività di ricerca nel mondo hardware correlate con questo progetto permetteranno ad esempio di migliorare la qualità delle immagini dei sensori Aptina in condizioni di illuminazione sfavorevoli o di introdurre un'unità sensoriale (IMU, composta da accelerometro, giroscopio e magnetometro), con cui ricavare ulteriori informazioni dall'ambiente circostante.

Per quanto riguarda la parte software, l'obiettivo in prima battuta sarà garantire una stretta interazione tra l'FPGA e la CPU. Sono in corso altri progetti di ricerca per l'installazione e la gestione di un sistema Linux e per permettere al sistema operativo di comunicare con l'hardware realizzato su logica programmabile. Dall'altra parte verrà successivamente effettuato il porting del sistema su scheda MicroZed; questo chip rappresenta di fatto una versione più piccola della ZedBoard, in quanto contiene lo stessa versione del SoC Zynq ma è dotata delle sole interfacce essenziali. Lo scopo ultimo di queste ricerche è quindi creare un sistema autocontenuto, indipendente e facilmente trasportabile (o indossabile) che permetta di svolgere elaborazioni di dati 3D senza l'intervento di dispositivi esterni.

References

[1] Stefano Mattoccia, *Stereo vision: algorithms and applications*, 2011

[2] Stefano Mattoccia, *Stereo vision algorithms on FPGAs*, 8th IEEE Embedded Vision Workshop,CVPR 2013,Portland,Oregon,USA,June 24, 2013

[3] Lorenzo Ciotti, *Sperimentazioni di metodologie per il trasferimento di dati da FPGA a DDR*, Tesi di Laurea, Università di Bologna, AA 2014-2015

[4] Alessandro Mattei, *Interfacciamento di sensori d'immagine stereo mediante connessione differenziale su piattaforma FPGA*, Tesi di Laurea, Università di Bologna, AA 2014-2015

[5] Vitaliano Gobbi, *Progetto e sperimentazione di un convertitore seriale parallelo su FPGA per un flusso video LVDS a elevata frequenza generato da una telecamera stereo*, Tesi di Laurea, Università di Bologna, AA 2010-2011

[6] Simone Mingarelli, *Streaming di immagini via ethernet con Zynq con sistemi operativi Standalone e Linux*, Tesi di Laurea, Università di Bologna, AA 2015-2016

[7] Elisa Rimondi, Pierlugi Zama Ramirez, Federica Albertini, *Sostituzione dell'IpCore Frame Buffer*, Relazione del progetto svolto nel corso di Logiche Riconfigurabili M, Università di Bologna, AA 2015-2016

[8] Silvio Olivastri, Cosimo Damiano Scarcella, Francesco Ricci, *Modulo di interfacciamento per sensori di immagine digitali (OV7670)*, Relazione del progetto svolto nel corso di Logiche Riconfigurabili M, Università di Bologna, AA 2015-2016 *Relazione OV*

[9] S. Mattoccia, M. Poggi, *A passive RGBD sensor for accurate and real-time depth sensing self-contained into an FPGA*, 9th ACM/SIGBED International Conference on Distributed Smart Cameras (ICDSC 2015), September 8-11, 2015, Seville, Spain

[10] S. Mattoccia, I. Marchio, M. Casadio, *A compact 3D camera suited for mobile and embedded vision applications*, 4th IEEE Mobile Vision Workshop, CVPR 2014, Columbus, Ohio, USA, June 23, 2014

[11] A. Muscoloni, S. Mattoccia, *Real-time tracking with an embedded 3D camera with FPGA processing*, International Conference on 3D (IC3D), 9-10 Dec, 2014, Liège, Belgium

[12] M. Poggi, L. Nanni, S. Mattoccia, *Crosswalk recognition through point-cloud processing and deep-learning suited to a wearable mobility aid for the visually impaired*, Image-based Smart City Application (ISCA2015), September 8, 2015, Genova, Italy

[13] S. Mattoccia, P. Macrì, G. Parmigiani, G. Rizza, *A compact, lightweight and energy efficient system for autonomous navigation based on 3D vision*, 10th IEEE/ASME International Conference on Mechatronic and Embedded Systems and Applications (MESA 2014), September 10-12, 2014, Senigallia, Italy

[14] OmniVision, *OV7670/OV7171 CMOS VGA CameraChip with OmniPixel Technology*

[15] Aptina, *MT9V033 1/3 inch Wide-VGA CMOS digital sensors Data Sheet*

[16] Jasmina Vasiljevic, Paul Chow, *MPack: Global memory Optimization for Stream Applications in High-Level Synthesis*

[17] Jasmina Vasiljevic, Paul Chow, *Using Buffer-to-BRAM Mapping Approaches to Trade-off Throughput vs. Memory Use*

[18] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, Robert W. Stewart, *The Zynq Book: Embedded Processing with the ARM Cortex-A9 on the Xilinx Zynq-7000 All programmable SoC*, 1st edition, 2014

[19] Xilinx, *UG585 - Zynq-7000 All Programmable SoC: Technical Reference Manual*, February 23, 2015

[20] Xilinx, *ZedBoard Hardware User's Guide*, Version 1.1, August 1, 2012

[21] Xilinx, *UG901 - Vivado Design Suite User Guide*, November 18, 2015

[22] Xilinx, *UG901 - Vivado Design Suite User Guide*, `http://www.xilinx.com/support/documentation/sw_manuals/xilinx2013_1/ug901-vivado-synthesis.pdf`, April 10, 2013

[23] Xilinx, *UG901 - Vivado Design Suite User Guide*, http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_3/ug901-vivado-synthesis.pdf, September 30, 2015

[24] Xilinx, *UG902 - Vivado Design Suite User Guide*, http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf, May 30, 2014

[25] Xilinx, *UG1037 - AXI reference Guide*, http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/latest/ug1037-vivado-axi-reference-guide.pdf, June 24, 2015

[26] Xilinx, *XAPP745 - Processor Control of Vivado HLS Designs*, http://www.xilinx.com/support/documentation/application_notes/xapp745-processor-control-vhls.pdf, September 4, 2012

[27] Xilinx, *XAPP793 - Implementing Memory Structures for Video Processing in the Vivado HLS Tool*, http://www.xilinx.com/support/documentation/application_notes/xapp793-memory-structures-video-vivado-hls.pdf, September 20, 2012

[28] Xilinx, *XAPP1209 - Designing Protocol Processing Systems with Vivado High-Level Synthesis*, http://www.xilinx.com/support/documentation/application_notes/xapp1209-designing-protocol-processing-systems-hls.pdf, August 8, 2014

[29] Xilinx, *Advanced Synthesis Techniques*, http://www.xilinx.com/publications/prod_mktg/club_vivado/presentation-2015/paris/Xilinx-AdvancedSynthesis.pdf0, 2015

[30] Xilinx, *AXI4 Technical Seminar*

[31] Xilinx, *UG111 - Embedded System Tools Reference Guide*, http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/est_rm.pdf, September 16, 2009

[32] Mohammadsadegh Sadri, *What is ZYNQ? (Lesson 1)*, https://www.youtube.com/watch?v=zD-5LQCOelI

[33] Mohammadsadegh Sadri, *ZYNQ Training - Session 01 - What is AXI?*, https://www.youtube.com/watch?v=0Dt8rWJdiJo

[34] Mohammadsadegh Sadri, *ZYNQ Training - Session 02 - What is an AXI Interconnect?*, https://www.youtube.com/watch?v=BASCRxR2L-c

[35] Mohammadsadegh Sadri, *ZYNQ Training - Session 02 - AXI Stream Interface*, https://www.youtube.com/watch?v=qNQ3Q2GrIYg

[36] Mohammadsadegh Sadri, *AXI Memory Mapped Interfaces & Hardware Debugging in Vivado (Lesson 5)*, https://www.youtube.com/watch?v=sl8xCMu39Mk

[37] The Development Channel, *VIVADO HLS Training - Introduction #01*, https://www.youtube.com/watch?v=kgae3Wzqngs

[38] The Development Channel, *VIVADO HLS Training - Array as parameters #02*, https://www.youtube.com/watch?v=h3ZHp8NygGQ

[39] The Development Channel, *VIVADO HLS Training - Custom Size Variables or parameters #03*, https://www.youtube.com/watch?v=BXMbsL_v0sk

[40] The Development Channel, *VIVADO HLS Training - Math Library #04*, https://www.youtube.com/watch?v=lo4Old_TFZU

[41] The Development Channel, *VIVADO HLS Training - AXI Lite slave floating point #05*, https://www.youtube.com/watch?v=aDaJIRoTlzQ

[42] The Development Channel, *VIVADO HLS Training - BRAM interface #06*, https://www.youtube.com/watch?v=BUVbKonhc2w

[43] The Development Channel, *VIVADO HLS Training - AXI Stream interface #07*, https://www.youtube.com/watch?v=3So1DPe2_4s

[44] The Development Channel, *VIVADO HLS Training - AXI Master #08*, https://www.youtube.com/watch?v=vW4gYgxAVmI

[45] Xilinx, *Vivado HLS Technical Introduction*, https://www.youtube.com/watch?v=5lYq8_bqAcI