Realizzazione di un kernel FPGA per migliorare le prestazioni di un algoritmo di Visual Odometry

Davide Malvezzi, Jia Cheng Hu
{212555,218863}@studenti.unimore.it

Ottobre 2019

1 Visual Odometry

La *Visual Odometry* (VO) consiste nella stima dell'ego-motion di un agente, ovvero cercare di stimare la posizione e l'orientamento 3D di un agente all'interno di un ambiente usando una o più telecamere installate.
Il problema di recuperare la struttura dell'ambiente (o perlomeno un'idea di ciò che circonda l'agente) e le posizioni 3D a partire da immagini 2D, ordinate o non, fa parte di una branca della *Computer Vision* chiamata *Structure From Motion* (SFM). La VO è un caso particolare della SFM in cui si richiede che le immagini siano ordinate temporalmente e che vengano tipicamente processate in real-time, non appena vengono acquisite, mentre nel caso più generale vengono accumulate e processate anche successivamente. Si tratta di un problema simile alla *Simultaneous Localization And Mapping* (SLAM), ma quest'ultima si concentra maggiormente nella ricostruzione consistente e globale del percorso (si è interessati ad esempio a trovare le "Loop Closure"), mentre nella VO si è più interessati alla correttezza locale e quindi di rendere consistenti le informazioni acquisite rispetto gli ultimi frame.

1.1 Pipeline generale di Visual Odometry

In questo lavoro ci focalizziamo sulla *Monocular Visual Odometry*. Il problema consiste quindi nel ricostruire la propria traiettoria e l'ambiente circostante tramite una sequenza di singoli frame. Dopo una fase di inizializzazione, la pipeline della VO [7] si presenta come in Figura 1.
Nella fase *Image Sequence*, la cosa importante da considerare è che le immagini subiscono una fase di pre-processing a seconda della tipologia di telecamera usata per l'acquisizione, in modo tale da convertire i dati in un formato a cui siamo abituati, ovvero la rappresentazione matriciale o "piatta" dell'immagine. Ogni tipologia di telecamera ha associato un modello. Il più comune è il modello

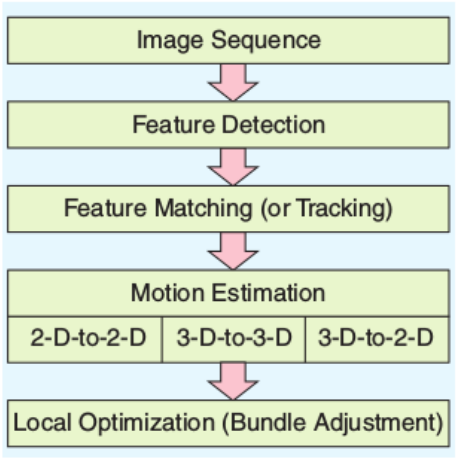

Figura 1: Pipeline generale della *Visual Odometry*.

PinHole che modella l'occhio umano. Altri due modelli famosi sono il modello *Spherical* e *Fish-Eye* che si differenziano per l'estensione visiva più ampia. Tipicamente, nei dati tecnici del sensore, vengono forniti anche dei parametri detti "intrinseci" che permettono di correggere via software la distorsione causata dalle lenti. Durante la fase di *Feature Detection* di ciascuna immagine si individuano un insieme di punti salienti, che sono tipicamente spigoli e bordi, tramite algoritmi di *Computer Vision* come SIFT e FAST. A questo punto, nella *Feature Tracking* avviene l'accoppiamento delle feature estrapolate dal frame corrente con quelle estratte in un frame precedente. L'obiettivo è quello di far sì che le feature siano consistenti in una diversa osservazione nel tempo dello stesso punto 3D nella mappa. Una volta collezionati un quantitativo sufficiente di punti accoppiati, si passa alla fase di *Motion Estimation*. Questa si occupa di estrapolare una matrice di trasformazione del corpo rigido (più formalmente $T \in SE(3)$) la quale racchiude il vettore di traslazione e la matrice di orientamento rispetto il punto di partenza. La sua accuratezza e la velocità con la quale viene calcolata dipendono profondamente dalla quantità e dalla qualità delle feature e dall'algoritmo utilizzato. Il concetto di qualità delle feature si riferisce non solo alla qualità visiva, ma anche alle coordinate del punto 3D a cui queste feature fanno riferimento. In particolare, fino ad ora non si è preso in considerazione la ricostruzione dell'ambiente circostante. Tuttavia nella VO, spesso si effettua il mapping a prescindere dal fatto che questo venga esplicitamente richiesto dal problema. Questo permette l'utilizzo del *Bundle Adjustment*, ovvero il meccanismo tramite il quale si cerca di correggere e minimizzare gli errori dovuti all'incertezza introdotta dai sensori, dalle approssimazioni e dal fenomeno di *Motion Drift*, errore nella traiettoria causato dagli errori accumulati nel tempo. Per fare ciò si eseguono due operazioni:la *Pose Graph Optimization* e la *Windowed Bundle Adjustment*. In breve, si cerca di rendere il più coerenti pos-

sibile le matrici di trasformazioni e i punti della mappa rispetto ad una finestra temporale degli ultimi frame acquisiti [8]. In questo processo, avere a disposizione una mappa diventa quindi di fondamentale importanza per migliorare la qualità della traiettoria ricostruita.

2 Introduzione a SVO

La pipeline specifica di VO che prenderemo in considerazione è denominata *Fast semi-direct monocular visual odometry* (SVO) [5]. Tuttavia, per comprendere il design di questo algoritmo, è doveroso precisare che prima della sua introduzione vi erano solamente due classi di algoritmi di VO:

- Feature based methods
- Direct methods

I metodi *Feature Based* effettuano una fase di *Feature Extraction* per ciascuna immagine in modo da individuare i punti salienti. Dopo di chè si effettua un matching/tracking di tali punti con le feature dell'immagine precedente secondo qualche criterio di similarità. Infine, la matrice di trasformazione e i punti nella mappa vengono calcolati tramite la geometria epipolare. Questo approccio risulta vincente grazie al suo basso costo computazionale e alla qualità della ricostruzione dovuta sopratutto all'utilizzo di buone tecniche per l'estrazione di punti salienti, presentando però carenze a livello di robustezza. Al contrario, i metodi diretti consistono nel dividere le immagini in regioni e calcolare la posizione della telecamera a partire direttamente dall'intensità dei pixel, minimizzando l'errore fotometrico su tali regioni tra un frame e l'altro. Questo risulta in un approccio più robusto rispetto al primo metodo, in certe circostanze come quelle in cui vi è mancanza di texture particolari (situazioni in cui le tecniche di feature extraction non forniscono buoni risultati) e al blurring. Risulta invece più pesante dal punto di vista computazionale.

2.1 SVO

SVO è un algoritmo di VO che rappresenta un approccio ibrido tra metodi diretti e feature based, cercando di estrapolare i punti di forza di entrambe le metodologie e di risolvere contemporaneamente loro i difetti. L'algoritmo si divide in due macro moduli, il primo si divide a sua volta in micromoduli e questi si relazionano secondo lo schema riportato in Figura 2:

- Modulo di Motion Estimation, che si divide nella Sparse Model-based Image Alignment, nella Feature Alignment e nella Pose & Structure Refinement
- Modulo di Mapping

I due moduli sono eseguiti in parallelo, ma ai fini della comprensione è meglio trascurare, al momento, tale aspetto.

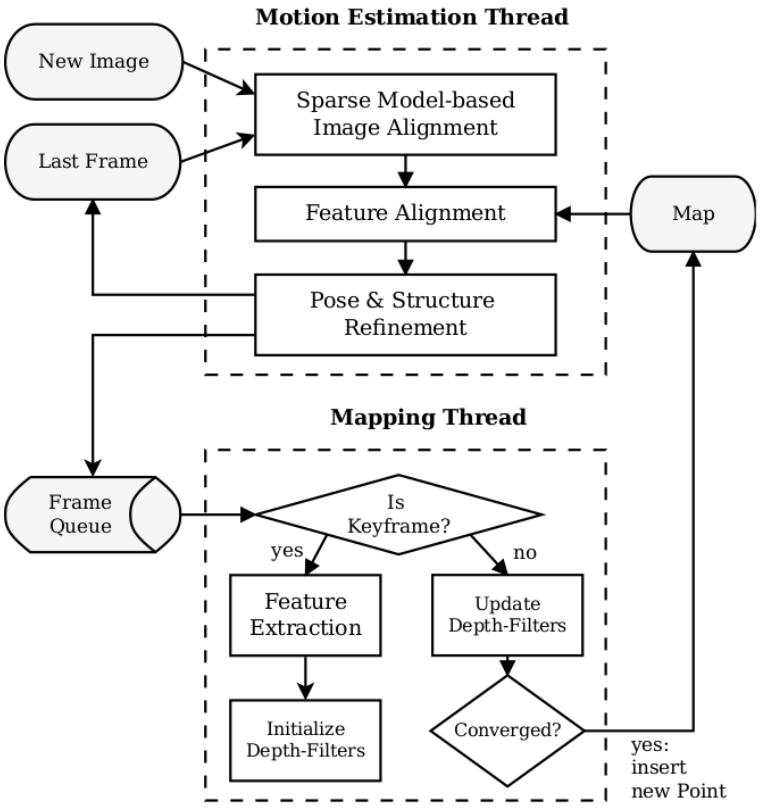

Figura 2: Schema di SVO.

Il modulo di Mapping implementa un modello per ricostruire la scenografia circostante. Mentre la Motion Estimation si occupa di recuperare la traiettoria e di fare bundle-adjustment. Quest'ultimo step dipende dalla mappa ricostruita dall'altro modulo. Si fornisce ora una descrizione qualitativa dell'algoritmo, lasciando i dettagli alle sezioni dopo. Le definizioni chiave per la comprensione dell'algoritmo sono le seguenti:

- Un *frame* è un immagine rappresentata come una matrice 2D di pixel.
- Un *keyframe* è un frame che si distingue dagli altri per aver contribuito nell'inserire nuovi punti nella mappa.
- La mappa è la collezione di punti 3D che costituiscono la ricostruzione della scenografia attraversata.
- Una *feature point* è punto 2D ritenuto saliente individuato in un frame.
- Una *feature patch* o semplicemente *patch*, è una matrice 2D dimensione contenuta (es. 8×8) incentrato in una feature point all'interno di un frame.
- La *feature extraction* è il processo tramite nel quale si estrapolano le feature point da un frame.
- La matrice di trasformazione del corpo rigido $C_k \in SE(3)$ è la matrice 4×4 che esprime l'orientamento e la posizione assoluta della telecamera nell'istante k, ovvero quella relativa al sistema di riferimento $(0, 0, 0)$ relativo alla posizione iniziale della telecamera.
- La matrice di trasformazione del corpo rigido $T_{k,k-1} \in SE(3)$ è la matrice 4×4 che esprime il cambiamento in termini di traslazione e orientamento della telecamera dal frame k-esimo (o del frame al tempo k-esimo) e quello precedente (k-1)-esimo.

L'obiettivo dell'algoritmo è calcolare la traiettoria della telecamera e ricostruire la scena attraversata. Grazie alla proprietà di concatenazione delle matrici di trasformazione (1) questo si traduce nel ricavare ad ogni step k la matrice di trasformazione $T_{k-1,k}$ ottenendo così la sequenza $T_{1,2}, T_{2,3}, T_{3,4}, ..., T_{k,k-1}$ e riempire la mappa con punti 3D più accurati possibili.

$$C_k = T_{1,2} * T_{2,3} * T_{3,4} * ... * T_{k-1,k} = C_k - 1 * T_{k-1,k} \tag{1}$$

Prima della pipeline di SVO avviene una fase di inizializzazione nella quale si individuano due keyframe da cui ottenere i primi punti nella mappa e la prima matrice di trasformazione $T_{1,2}$. Al generico passo k la fase di Sparse Image Alignment calcola, a partire dalla matrice di trasformazione precedente $T_{k-2,k-1}$ e i punti 3D nella mappa, i punti visibili anche nel nuovo frame e trova la nuova matrice di trasformazione $T_{k-1,k}$ tramite un algoritmo iterativo. Allo stesso tempo, vengono individuati dei nuovi punti di feature points senza

passare per la feature extraction. Durante la fase di feature alignment ulteriori feature point vengono aggiunti e aggiustati per renderli consistenti rispetto la mappa. Nella fase di Pose & Structure Refinement, si ottimizza la nuova matrice di trasformazione tramite la nuova posizione di tutte le feature point e si cerca di rendere consistente i punti della mappa rispetto alla traiettoria.
Il modulo di Depth estimation è quello che si occupa di inserire nuovi punti nella mappa. A parte l'inizializzazione, in cui i primi punti vengono calcolati tramite la geometria epipolare, in SVO questi vengono aggiunti in base alla selezione dei keyframe. Dall'inizializzazione vengono definiti due keyframe, quindi è fornito il concetto di ultimo keyframe. Se il nuovo frame acquisito soddisfa certe condizioni viene definito un nuovo keyframe e di conseguenza viene effettuato una feature extraction, e per ciascun punto saliente estratto viene inizializzato un cosidetto depth filter, che altro non è che un modello probabilistico per stimare con una accuratezza tanto più alta quanto i frame (non keyframe) utilizzati a stimarla. E una volta che tale modello riporta un punto con una certa sicurezza, questo viene inserito nella mappa.
Il tutto potrebbe avvenire in parallelo, in quanto il calcolo e l'aggiornamento dei depth filter risulta computazionalmente molto più veloce di tutta la fase di motion estimation.
In seguito, i primi 3 sottoparagrafi descrivono la *PIPELINE* di SVO, infatti ogni step, fa uso del risultato dello step precedente.

2.1.1 Sparse Image Alignment

L'algoritmo di Sparse Image Alignment consiste in un problema ai minimi quadrati che viene risolto tramite l'algoritmo iterativo di Gauss-Newton.

$$T_{k,k-1} = \arg\min_{T_{k,k-1}} \frac{1}{2} \sum_{i \in \bar{R}} \|\delta I(T_{k,k-1}, u_i)\|^2 \tag{2}$$

In sostanza, nella formula (2) si vuole cercare la matrice di trasformazione $T_{k,k-1}$ che minimizzi la differenza fotometrica di tutte le patch che si possono osservare sia nel frame corrente k che in quello precedente $k-1$ (questo è l'insieme dei punti $\bar{R}$). Per fare ciò la matrice $T_{k,k-1}$ dovrà esser tale da far assomigliare le patch attorno ai feature point, individuati nel frame precedente, con le patch attorno ai nuovi feature point nel nuovo frame ottenuti tramite riproiezione dei punti 3D a cui i feature point del frame precedente facevano riferimento (le formule di quanto descritto è omessa e rappresentata da δI). Se la matrice $T_{k,k-1}$ è ottimale, la differenza fotometrica delle patch è zero, in quanto l'orientamento risulta far combaciare tutte le patch del frame precedente con quello corrente, prevedendo quindi esattamente lo spostamento effettuato dalla telecamera.

In caso contrario se la matrice $T_{k,k-1}$ non fosse adatta, le patch ottenute tramite riproiezione dei punti nel nuovo frame sarebbero posizionati male rispetto alla vera collocazione delle feature, rendendo alto l'errore fotometrico con le patch del frame precedente.

Un aspetto molto importante, che motiva il successo di SVO, è che in questa fase vengono risolti i difetti dei metodi diretti e feature based, pur essendo un ibrido dei due:

- si scelgono tante patch, ma di dimensioni ridotte, a differenza dei metodi diretti
- non viene effettuato la feature extraction, a differenza dei metodi feature-based
- come conseguenza dei due punti precedenti SVO ha una parte di Motion Estimation molto leggero computazionalmente

Si nota infatti che l'algoritmo non effettua mai nessuna Feature Extraction in questa fase, in particolare l'algoritmo non ha idea di dove si trovino le feature nel nuovo frame, ma questo aspetto viene risolto grazie al fatto che, definito la matrice di trasformazione, vengono definiti in maniera automatica anche delle feature point nel nuovo frame. Rimane però compito dell'ottimizzatore trovare la matrice $T_{k,k-1}$ tale per cui queste feature point siano effettivamente delle feature point.
L'algoritmo è di tipo iterativo, e quindi fa utilizzo del concetto di gradiente, uno strumento di ricerca di minimi locali. Ma le immagini non hanno una superficie regolare, semplice e con un unico minimo globale. É naturale chiedersi se questo non causi problemi. In verità, la risposta è NO, in quanto inizializzando la matrice $T_{k,k-1}$ al punto $T_{k-1,k-2}$ si è abbastanza sicuri di trovarsi in una regione vicino al minimo globale.

2.1.2 Feature Alignment

L'operazione di Sparse Image Aligment restituisce la matrice di trasformazione corrente. A partire da questo è possibile individuare tutti i punti nella mappa che sono visibili da questa nuova posizione. Questo permmette di individuare tutte le feature point, senza utilizzare un metodo di *Feature Extraction* costituendo uno dei punti di forza di SVO (sebbene comunque, in questa fase non vengano ritrovati feature point nuovi). In questa fase tutti i punti 3D visibili dalla nuova posizione vengono proiettate nel frame corrente e contemporaneamente corrette e ottimizzate seguendo la seguente formulazione:

$$u_i^{'} = \arg\min_{u_i^{'}} \frac{1}{2} \left\| I_k(u_i^{'}) - A_i \dot{I}_r(u_i) \right\|^2, \forall i \tag{3}$$

Per ogni punto i visibile nella mappa viene individuato un keyframe di riferimento (denotato con r) il cui angolo di osservazione è il più assomigliante possibile al frame corrente. Successivamente tale punto viene proiettato nel frame corrente e nel frame di riferimento individuando rispettivamente le feature point $u_i^{'}$ e u_i. A questo punto l'ottimizzazione consiste nel minimizzare l'errore fotometrico di una patch incentrata nella feature point corrente $I_k(u_i^{'})$ e la patch corrispondente nel frame di riferimento $I_r(u_i)$ tramite un approccio iterativo basato su

discesa del gradiente (più dettagliamente fa uso dell'algoritmo di Lucas-Kanade, che a sua volta fa uso della procedura iterativa Gauss-Newton).

2.1.3 Pose & Structure Refinement

Lo step di Sparse Image Alignment non tiene conto delle feature point che non sono condivise tra il frame corrente e quello precedente e per questo motivo deve essere corretto tramite l'operazione di Pose Refinement.
La Pose Refinement consiste nella correzione della matrice di trasformazione appena ottenuta sfruttando le informazioni fornite dalla mappa. In particolare, l'operazione viene espressa dalla seguente minimizzazione:

$$T_{k,w} = \arg\min_{T_{k,w}} \frac{1}{2} \sum_i \parallel u_i - \pi(T_{k,w},\ {}_wp_i) \parallel^2 \tag{4}$$

La matrice di trasformazione $T_{k,w}$ è la matrice che fa passare dalle coordinate globali (world) a quello della telecamera, ed è quindi la matrice che effettivamente descrive la posizione della telecamera. Questo è calcolato tramite la $T_{k,k-1}$ ottenuta nella Sparse Image Alignment, la quale è soggetta maggiormente ad errori. Per questo viene effettuato questo step di refinement in cui la matrice $T_{k,w}$ viene ottimizzata affinchè minimizzi la distanza in norma quadratica delle feature point u_i ottenuto dallo step precedente, e la posizione della proiezione del punto stesso p_i (denotato da $\pi(T_{k,w},\ {}_wp_i)$) sul frame corrente, nel caso questo fosse descritto dalla matrice $T_{k,w}$ scelta.
Notiamo che i punti ottenuti nello step precedente corrispondono alla proiezione di tutti i punti visibili della mappa per cui si forza la coerenza della matrice di trasformazione con la mappa calcolata fino al momento corrente.
La Structure Refinement consiste nello step contrario, ovvero si aggiustano i punti della mappa tramite la *reprojection error minimization*. Si considerano gli ultimi n frames, e per ogni frame si minimizza la distanza tra il punto ${}_wp_i$ della mappa e il feature point associato del frame k-esimo riproiettato nella mappa.

$$\arg\min_{{}_wp_i} \sum_i \sum_{k \in Window} \parallel {}_wp_i - \pi^{-1}(u_i^k) \parallel^2 \tag{5}$$

La π^{-1} è l'operazione di riproiezione che parte da una feature point di un frame e riporta ad un punto della mappa. La procedura di Gauss-Newton è quella che effettivamente si occupa di trovare la nuova posizione delle feature point

2.1.4 Depth Filter

Tutto il processo di Mapping ruota attorno alla seguente formula:

$$p(\tilde{d_i^k} \| d_i, \rho_i) = \rho_i \mathcal{N}(\tilde{d_i^k} \| d_i, \tau_i^2) + (1 - \rho_i)\mathcal{U}(\tilde{d_i^k} \| d_i^{min}, d_i^{max}) \tag{6}$$

Viene modellato tramite una distribuzione Gaussiana + Uniforme. Tale gaussiana, immaginiamo di modellarla avente come media d_i e varianza τ_i^2, dove d_i rappresenta la distanza effettiva tra il punto 2D sull'immagine e il punto 3D

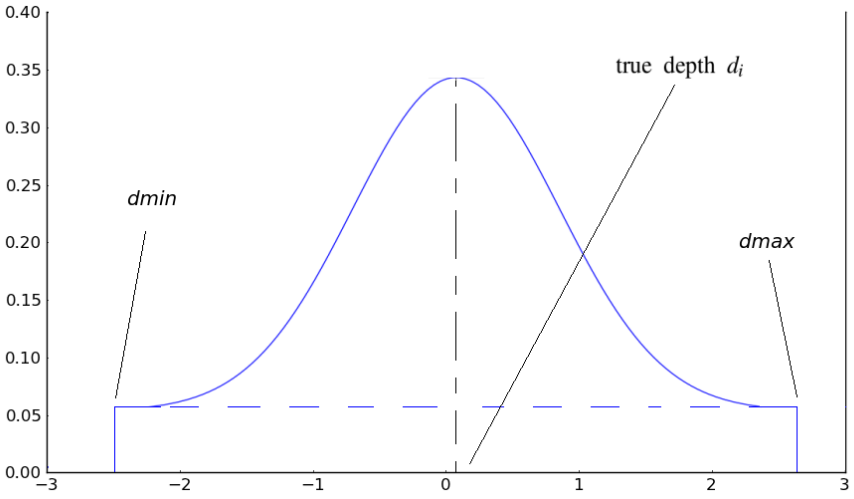

Figura 3: Distribuzione gaussiana + uniforme. Nell'asse-x si trova la distanza, mentre nell'asse y abbiamo la probabilità

e τ_i^2 l'incertezza relativa a tale distanza. La distribuzione uniforme serve per porre un intervallo d_{min} e d_{max} oltre i quali è 0 (vedi Figura 3), ma serve anche per rappresentare anche la probabilità degli outlier per distinguere tra outlier e inlier consideriamo che gli outlier abbiano la probabilità dell'uniforme (1-p_i) mentre gli inlier hanno quello della gaussiana.
I frame k per cui $k > r$ dove r è l'ultimo keyframe di riferimento vengono utilizzati per aggiornare d_i e ridurre la varianza τ_i^2 del depth filter, che è la metrica di affidabilità del modello, o la bontà della stima del punto 3D. L'aggiornamento continua finchè la varianza non raggiunge una soglia sufficientemente bassa da poter essere inserita nella mappa.
L'aspetto di SVO che rende questa pipeline di visual odometry tanto efficiente è il fatto che non vengono eseguiti algoritmi di Feature Extraction (ad esempio SIFT o FAST) ad ogni frame, ma solamente quando si identifica un key-frame. Questo viene utilizzato per inizializzare i depth filter, ovvero i punti che verranno poi aggiunti nella mappa a seguito della convergenza del modello matematico descritto in questo paragrafo associato a ciascuno di essi.
Il modello matematico viene proposto perchè non essendo in contesti ideali, non è possibile ottenere la posizione effettiva di un punto 3D tramite semplicemente l'intersezione delle rette proiettate a partire da due frame con due feature che si riferiscono allo stesso punto. Quello che invece è possibile ottenere da un frame all'altro è una stima, e tale stima della distanza del punto 3D si suppone provenga da una distribuzione come descritta sopra. Ogni volta che si effettua una stima della distanza a partire da un frame $k > r$ allora è come ottenere un campione da questa distribuzione e una volta che si hanno abbastanza campioni,

si può inferire i parametri del modello che l'hanno generato. Nel nostro caso siamo interessati alla media, che è l'unica informazione che mancava al keyframe per ottenere le vere coordinate del punto 3D.

2.2 Profilazione

Per il suo corretto funzionamento, SVO richiede molti parametri precisi di configurazione che variano a seconda dei contesti applicativi in cui è utilizzato, come ad esempio l'angolazione della telecamera (forward o downward motion), velocità del dispositivo in movimento (es. drone che ruota in modo lento e automobile che percorre strade). Per questo motivo, per fare un profiling si sono provati molti dataset e diverse configurazioni in modo da trovare un caso in cui l'algoritmo funzionasse in condizioni ideali e il carico di lavoro fosse tale da giustificare un'eventuale parallelizzazione.
Si sono provati diversi dataset e quelli per cui si è trovato una configurazione soddisfacente sono Machine Hall 01, Vicon Room 01 della EuRoC MAV Dataset [4] e AirGround fornito dal paper del codice sorgente di SVO [6]. Il dataset con il quale si è ottenuto il risultato più soddisfacente in termini di carico computazionale e accuratezza della traiettoria è il Machine Hall Easy 01 (Figura 5). Per testarli si è utilizzato il software ROS [1], un framework per lo sviluppo e testing di programmi destinati alla robotica. La piattaforma di testing è la board ZCU102 della Xilinx, con una Zynq UltraScale+ MPSoC che dispone di una parte di CPU o logica di controllo ARM A53 e una parte di logica programmabile FPGA. Si è scelto di installare Ubuntu 16.04 come sistema operativo della board.
Il profiling è stato effettuato con il tool Oprofile [2] e i risultati ottenuti sono riportati nella Tabella (1). Si può osservare che le funzioni più onerose sono la trasformazione affine, e lo step di Feature Alignment. Ci focalizzeremo in seguito a velocizzare tramite FPGA il secondo step, in quanto costituisce un operazione più caratteristica di SVO rispetto la trasformazione affine che è un operazione generica di una pipeline di VO. Si nota inoltre come la parte di estrazione delle features si trovi in ottava posizione con un contributo trascurabile in quanto effettuato solamente all'identificazione di una keyframe, confermando la caratteristica ibrida di SVO.

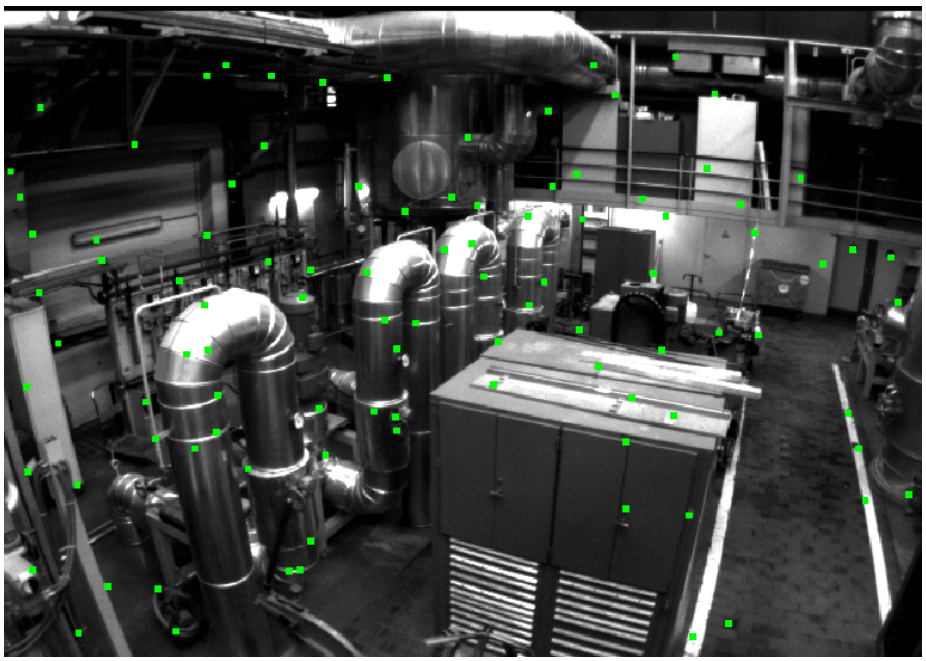

Figura 4: Frame tipico del dataset Machine Hall 01, in verde sono indicati le feature point individuate.

Tabella 1: Le prime 10 funzioni più costose

Pos.	N.Calls	Contributo (%)	Nome del simbolo
1	242269	12.1038	svo::warp::warpAffine
2	237907	11.8859	svo::feature_alignment::align2D
3	214266	10.7048	/lib/aarch64-linux-gnu/libpng12.so.0.54.0
4	147312	7.3598	/usr/lib/aarch64-linux-gnu/libopencv_core.so.2.4.9
5	137398	6.8644	..Eigen..
6	80872	4.0404	/lib/aarch64-linux-gnu/libz.so.1.2.8
7	71701	3.5822	svo::Matcher::findEpipolarMatchDirect
8	59772	2.9862	fast::fast_corner_detect_10
9	57035	2.8495	svo::SparseImgAlign::computeResiduals
10	56847	2.8401	Sophus::SO3::operator*(Eigen::Matrix...)

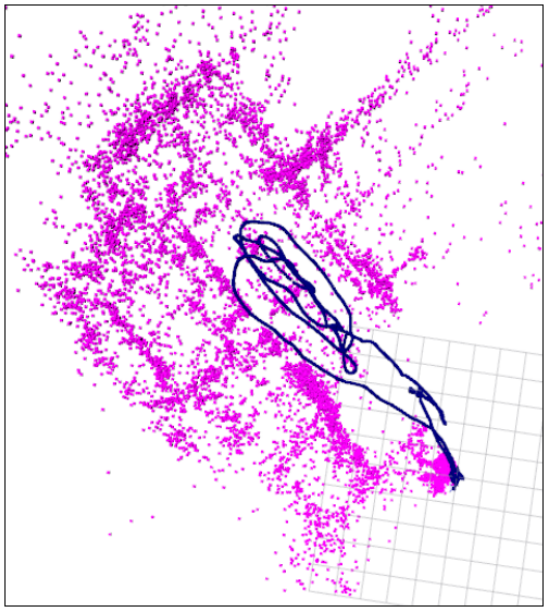

Figura 5: Traiettoria ricostruita (linea blu) e i punti della mappa (punti viola) osservata dall'alto.

3 Kernel *BatchAlign2D*

Il metodo denominato *Align2D* riceve in input il frame corrente I_t, un insieme di patch P_i 8×8 con relativo borda da 1 pixel e le corrispondenti posizioni $u_i = (x_i, y_i)$ in pixel. L'obiettivo del metodo è quello di modificare le posizioni delle patch in modo da minimizzare l'errore fotometrico tra il frame I_t e le patch. Questo problema è risolto utilizzando l'algoritmo *inverse compositional Lucas-Kanade* [3]. L'algoritmo utilizza il metodo iterativo *Gauss-Newton* per calcolare una sequenza di spostamenti ϵ_k che permettono di muovere ogni patch in una posizione che minimizza l'errore complessivo.
Il metodo *Align2D* può essere diviso in due fasi:

- una fase di inizializzazione in cui vengono calcolate quantità costanti
- una fase di ottimizzazione in cui si applica *Gauss-Newton*

Input : Il frame corrente I_t
Una patch P 8×8 con relativo bordo di 1 pixel
La posizione iniziale u della patch P
Output: La nuova posizione u' della patch P
Procedure *Align2D*(I_t, P, u)
$J, H^{-1} =$ *initialization*(P);
$u' =$ *Gauss-Newton*$(I_t,\ P,\ u,\ J,\ H^{-1})$;
return u';

Algorithm 1: Metodo *Align2D*

L'ottimizzazione compiuta su ogni patch è indipendente da quella compiuta sulle altre, questo permette quindi di eseguire in parallelo l'ottimizzazione su ogni singola patch. Si vuole perciò realizzare un *kernel* FPGA in modo da poter fare l'*offloading* di un batch di patch ed eseguire concorrentemente la minimizzazione dell'errore fotometrico per ognuna di esse.

3.1 Inizializzazione

Nella fase di inizializzazione vengono calcolati i gradienti della patch P come

$$J(x,y) = \left[\frac{dP}{dx}(x,y), \frac{dP}{dy}(x,y), 1\right]$$

dove

$$\frac{dP}{dx}(x,y) = \frac{P(x+1,y) - P(x-1,y)}{2}$$
$$\frac{dP}{dy}(x,y) = \frac{P(x,y+1) - P(x,y-1)}{2}$$

Questi vettori sono salvati nell'insieme $J = \{J(x,y), x = 1 \dots 8, y = 1 \dots 8\}$ che verrà utilizzato successivamente dal metodo *Gauss-Newton*. I vettori $J(x,y)$ sono poi utilizzati per calcolare la matrice Hessiana H come

$$H = \sum_{(x,y)\in P} J(x,y)J(x,y)^T$$

L'inizializzazione ritorna l'insieme J dei gradienti e l'inversa della matrice Hessiana H^{-1}.

Input : Una patch P 8x8 con relativo bordo di 1 pixel
Output: Insieme dei gradienti J
Inversa della matrice Hessiana H^{-1}
Procedure *initialization(P)*
$J = \{\}$;
$H = zeros(3, 3)$;
for y = *1...8* **do**
for x = *1...8* **do**
$J(x, y) = \left[\frac{P(x+1,y)-P(x-1,y)}{2}, \frac{P(x,y+1)-P(x,y-1)}{2}, 1\right]$;
$J = J \cup J(x, y)$;
$H = H + J(x, y)J(x, y)^T$;
end
end
return J, H^{-1};

Algorithm 2: Inizializzazione del metodo *Align2D*

3.1.1 Ottimizzazione delle operazioni aritmetiche

Alcuni accorgimenti possono essere fatti per ridurre il numero di operazioni aritmetiche svolte in questa fase. Per semplicità di notazione indichiamo con $J_k = [dx_k, dy_k, 1]$ un elemento dell'insieme J.
Dalla definizione precedente della matrice Hessiana H

$$\begin{aligned} H &= \sum_{(x,y)\in P} J(x, y)J(x, y)^T \\ &= \sum_{k\in|J|} J_k J_k^T \\ &= \sum_{k\in|J|} \begin{bmatrix} dx_k \\ dy_k \\ 1 \end{bmatrix} \begin{bmatrix} dx_k & dy_k & 1 \end{bmatrix} \\ &= \sum_{k\in|J|} \begin{bmatrix} dx_k^2 & dx_k dy_k & dx_k \\ dx_k dy_k & dy_k^2 & dy_k \\ dx_k & dy_k & 1 \end{bmatrix} \end{aligned}$$

Si nota innanzitutto che la matrice H è simmetrica, possiamo quindi calcolare solamente i valori della matrice triangolare superiore. Inoltre, portando la sommatoria su k dentro la matrice, l'elemento $H[3, 3]$ risulta costante e pari a $|J| = 64$. Infine, nel calcolo degli elementi dx_k e dy_k è sempre presente una divisione per 2. Possiamo rimuovere questa operazione dal calcolo di $J(x, y)$ e applicarla una sola volta quando il calcolo della matrice H è stato completato. Questo permette di sostituire 128 divisioni *floating point* con solamente 5 operazioni equivalenti.
L'inizializzazione può essere quindi riscritta come

Input : Una patch P 8x8 con relativo bordo di 1 pixel
Output: Insieme dei gradienti $\overline{J}$ moltiplicati per 2
Inversa della matrice Hessiana H^{-1}
Procedure *initialization(P)*

$\overline{J} = \{\}$;
$H = zeros(3, 3)$;
for y = *1...8* **do**
  for x = *1...8* **do**
    $\overline{dx} = P(x+1, y) - P(x-1, y)$;
    $\overline{dy} = P(x, y+1) - P(x, y-1)$;
    $\overline{J} = \overline{J} \cup [\overline{dx}, \overline{dy}]$;
    $H[1,1] = H[1,1] + \overline{dx}^2$;
    $H[1,2] = H[1,2] + \overline{dx} \cdot \overline{dy}$;
    $H[1,3] = H[1,3] + \overline{dx}$;
    $H[2,2] = H[2,2] + \overline{dy}^2$;
    $H[2,3] = H[2,3] + \overline{dy}$;
  end
end
$H[1,1] = H[1,1]/4$;
$H[1,2] = H[1,2]/4$;
$H[1,3] = H[1,3]/2$;
$H[2,1] = H[1,2]$;
$H[2,2] = H[2,2]/4$;
$H[2,3] = H[2,3]/2$;
$H[3,1] = H[1,3]$;
$H[3,2] = H[2,3]$;
$H[3,3] = 64$;
return $\overline{J}, H^{-1}$;

Algorithm 3: Inizializzazione ottimizzata del metodo *Align2D*

Essendo la matrice H 3x3, la sua inversa H^{-1} può essere calcolata semplicemente con il metodo dei cofattori [9].

3.1.2 Ottimizzazione *HLS*

Partendo dalla versione ottimizzata dell'inizializzazione possiamo ora inserire le direttive *HLS* per parallelizzare questa porzione del metodo *Align2D*. Per prima cosa è possibile collassare i due cicli in uno unico e applicare la direttiva **PIPELINE**. Questa direttiva permette di creare una pipeline per ridurre la latenza dell'intero ciclo. Per massimizzare il parallelismo possiamo inoltre utilizzare la direttiva **ARRAY_PARTITION** sulle variabili P, J, H e H^{-1}. In questo modo i loro elementi vengono suddivisi su *BRAM* separate, così da poter accedere a ciascuno di essi in parallelo.

3.2 Ottimizzazione della Gauss-Newton

La procedura di Gauss-Newton è quella che effettivamente si occupa di trovare la nuova posizione delle feature point. La procedura fa utilizzo del gradiente e dell'inversa della matrice hessiana calcolati durante l'inizializzazione. Questi valori rimangono immutati e vengono utilizzati durante tutti gli step iterativi, in quanto sono relativi al mini-patch del frame di riferimento r. Questi vengono combinati con la parte della formula del gradiente che invece dipende dalla patch attorno alla feature point sul frame corrente k e la patch va ripetutamente letta in quanto ad ogni step la feature point cambia posizione e così anche il contenuto e il gradiente rispetto la nuova patch.
L'implementazione più semplice consiste nel trasferire tutta l'immagine e considerare le patch attorno alle feature point. Questi dati vengono trasferiti nella memoria più capiente dell'FPGA: le *Block RAM*. Questa tipologia di memoria presenta però il difetto non poter fisicamente permettere più di 2 accessi in parallelo. Questo implica che affinchè vengano processati più di 2 feature alla volta, sia necessario trasferire la stessa immagine più volte, abbattendo così enormemente le prestazioni.
Per risolvere questo problema, l'idea è quella di estrapolare e utilizzare solamente le *regioni* di immagini necessarie alla feature point per spostarsi nell'immagine originale. In particolare ciascuna iterazione fa uso delle informazioni contenute in una mini-patch di dimensione 8×8 intorno alla feature point per spostarsi tramite discesa del gradiente (vedi Figure 6), quindi la dimensione della regione deve considerare quanto al massimo la feature point può spostarsi nell'algoritmo originale. Una regione che risulta idonea per ottenere gli stessi risultati di SVO originale è una regione 64×64. Tuttavia, la diminuzione delle dimensioni di questa regione permette di aumentare ulteriormente le performance e può anche essere sfruttata come forma di regolarizzazione, siccome l'algoritmo di SVO non impone una condizione di terminazione precisa. Si può quindi configurare questo parametro a dimensioni minori quali ad esempio 32×32, 16×16. Accade spesso poi che la maggior parte delle feature point dopo la correzione si siano spostate di pochi pixel e quindi regioni di dimensioni minori permetto di ottenere gli stessi risultati. L'idea della regolarizzazione è di fermare l'iterazione di Gauss-Newton nel momento in cui la nuova posizione della feature tenta di uscire dalla regione. La dimensione scelta nei nostri esperimenti è di 64×64, optando così per la totale conformità con l'algoritmo originale.
Essendo le regioni molto più piccole dell'intera immagine originale (64×64 contro 752×480) è praticabile ora trasferire gruppi di feature point in block ram così da essere accedibili singolarmente da ciascun modulo di Gauss-Newton. In questo modo si otteniene il massimo parallelismo possibile. In particolare, si considera che tutte le feature point provengano da una stessa immagine e l'algoritmo consiste nel estrapolare tutte le regioni, una per ogni feature. Denotiamo la funzione di estrapolazione $batchPrepareRegion$ la quale implementazione consiste semplicemente nel prendere il riquadro di pixel nell'immagine originale centrato nella posizione della feature point e facendo attenzione ai casi particolari (come le feature point che si trovano vicino o sul bordo).

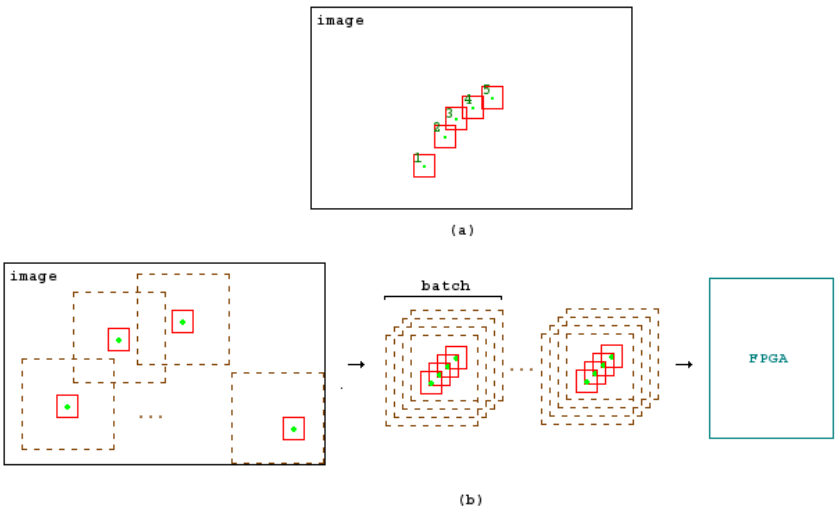

Figura 6: (a) L'immagine illustra una possibile sequenza di spostamenti di una feature point nella procedura di Gauss-Newton. Lo spostamento di una feature point termina quando si raggiunge il numero massimo di iterazioni prefissate o se la nuova posizione è al di fuori della regione estratta. (b) Questa parte illustra il concetto di batch, dove si considerano solamente B_{size} feature point alla volta. Per ogni feature point viene prima estratta la regione nel suo interno e poi vengono trasferite alla FPGA. I riquadri tratteggiati indicano la regione estrapolata e i puntini verdi all'interno il punto iniziale della feature.

Per aumentare ulteriormente le performance è possibile coprire i tempo di trasferimento della memoria tramite l'esecuzione *interleaved* tra FPGA e PS. Siccome la *batchPrepareRegion* avviene su PS e il kernel viene eseguito su FPGA, possiamo pensare di preparare il batch successivo mentre quello corrente sta venendo elaborato. Per fare questo definiamo la struttura dati *Batch* che contiene tutti i dati relativi ad un batch da trasferire dal PS alla FPGA. Per sfruttare l'*interleaving* possiamo creare due strutture *Batch* e mentre una sta venendo trasferita verso la FPGA possiamo applicare il metodo *batchPrepareRegion* sulla seconda. All'iterazione successiva sarà passata la struttura preparata all'iterazione precedente e si procederà a preparare la prima. Così per ogni iterazione.

Input : Immagine I, con N feature points
Output: Le nuovi posizione di feature points, corrette tramite Gauss-Newton
Procedure *DoubleBufferingFeatureAlignment(I)*

```
Batch[0] ← new Batch;
Batch[1] ← new Batch;
initialize(Batch[0]);
initialize(Batch[1]);
numBatches = ⌈N/B_size⌉;
for i = 0...numBatches-1 do
    previous_idx = (i − 1) % 2;
    curr_idx = i % 2;
    br ← batchPrepareRegion(I, from = i·B_size, to = (i+1)·B_size);
    transferData(br, Batch[curr_idx]);
    if i != 0 then
        waitUntilFPGAisDone();
        r ← getResultsFrom(Batch[previous_idx]);
        updateFeaturePoints(I, r)
    end
    startFPGA(Batch[curr_idx]);
    if i == numBatches-1 then
        waitUntilFPGAisDone();
        r ← getResultsFrom(Batch[curr_idx]);
        updateFeaturePoints(I, r)
    end
end
```

Algorithm 4: Implementazione con double buffering del *Align2D* processato in batches su FPGA

3.2.1 Ottimizzazione *HLS*

Innanzitutto, è possibile usare la direttiva **ARRAY_PARTITION** sulle regioni di immagini in modo da favorire l'accesso parallelo alla memoria. La direttiva **UNROLL** è poi utilizzata per far si che l'ottimizzazione di ogni singola patch avvenga in parallelo. Infine, all'interno dell'algoritmo iterativo si applica la direttiva **PIPELINE** per il calcolo dell'errore fotometrico.

3.3 Kernel FPGA

Il kernel FPGA *BatchAlign2D* riceverà in ingresso un batch di B_{size} patch e le relative informazioni per applicare il metodo *Align2D* in parallelo ad ognuna di queste. In particolare, i parametri di ingresso del metodo *BatchAlign2D* sono:

- un batch di region B_R
- un batch di patch B_P

- un batch di posizioni iniziali B_u
- il numero massimo di iterazioni it_{max} per il metodo *Gauss-Newton*

Mentre i valori di ritorno sono:

- un batch di posizioni finali $B_{u'}$
- un batch di flag B_c

Per ogni $k = 1...B_{size}$, si applica il metodo *Align2D* alla patch B_P^k con posizione iniziale B_u^k in modo da minimizzarne l'errore fotometrico con la region B_R^k. La posizione finale sarà salvata in $B_{u'}^k$, mentre il flag B_c^k indicherà se il metodo *Gauss-Newton* è arrivato a convergenza in it_{max} o meno iterazioni.
Per eseguire in parallelo l'ottimizzazione sulle singole patch è possibile utilizzare la direttiva *HLS* **UNROLL**.

Procedure *BatchAlign2D*(B_R, B_P, B_u, it_{max})
  for $k = 1...B_{size}$ **do**
    $\overline{J}, H^{-1} = initialization(B_P^k)$;
    $B_{u'}^k$, B_c^k = *Gauss-Newton*(B_R^k, B_P^k, B_u^k, it_{max}, $\overline{J}, H^{-1}$);
  end
  return $B_{u'}$, B_c;

Algorithm 5: Kernel *BatchAlign2D*

3.3.1 Architettura

La Figura 7 mostra una schematizzazione dell'architettura creata per la comunicazione tra PS e FPGA e l'architettura interna del kernel *BatchAlign2D*. Utilizzando lo strumento di sintesi di HLS è stato possibile studiare la quantità di risorse hardware utilizzate dal kernel al variare del numero di patch elaborate in parallelo. In Tabella 2 sono riportati i risultati ottenuti con i diversi valori di B_{size} che sono stati provati. Si può vedere dai dati raccolti che le percentuali di utilizzo delle risorse crescono linearmente all'aumentare di B_{size}. Questo andamento era atteso in quanto i blocchi hardware per elaborare le patch sono tutti identici tra di loro.

$\mathbf{B_{size}}$	**BRAM 18K**	**DSP48E**	**FF**	**LUT**	**URAM**
1	0	3	4	8	0
4	0	9	13	21	0
8	1	17	24	40	0
12	1	25	34	64	0
16	2	33	45	87	0

Tabella 2: Percentuali di risorse utilizzate al variare di B_{size}.

I tempi di comunicazione tra PS e FPGA sono un fattore importante da considerare quando si vuole fare l'*offloading* di un metodo, in quanto possono generare *overhead* ed avere un alto impatto sullo *speed-up* finale. Per evitare ciò si è

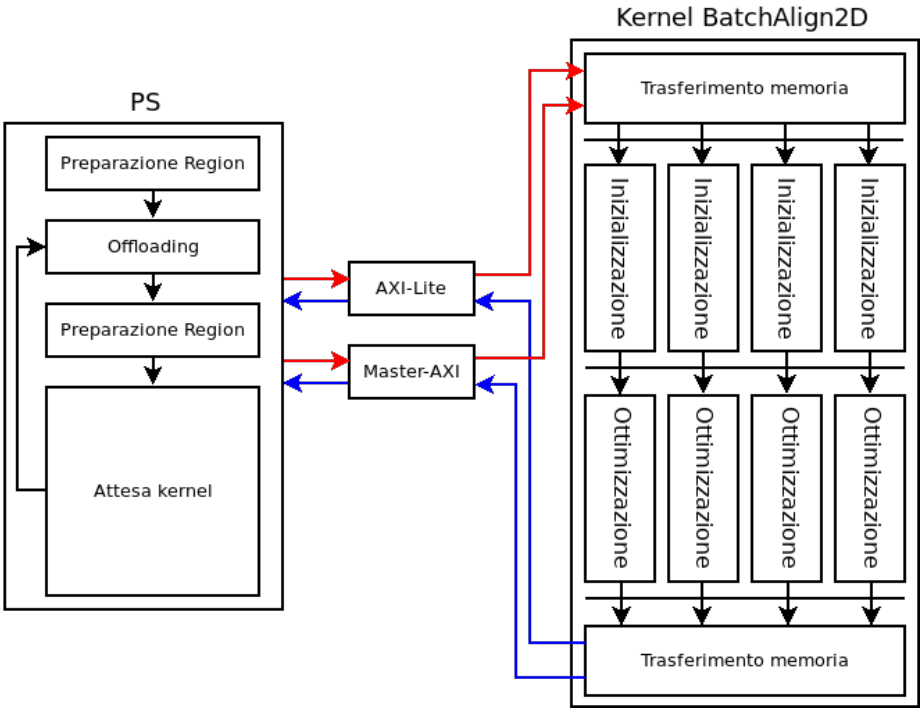

Figura 7: Schematizzazione dell'architettura interna del kernel *BatchAlign2D* e della comunicazione tra PS e FPGA.

cercato di ridurre al minimo la quantità di dati trasferiti e sono stati utilizzati protocolli di comunicazione che permettono il trasferimento di grandi quantità di dati a bassa latenza. A questo proposito, ogni region è stata vettorizzata e questi vettori sono stati concatenati in unico vettore i cui elementi sono interi senza segno a 8 bit essendo le immagini in scala di grigi. Analogamente, tutti i batch sono stati vettorizzati. Questo permette di sfruttare la località spaziale dei dati per effettuare trasferimenti in *burst*. La dimensione di B_c può essere ridotta a B_{size} bit utilizzando solamente un bit per ciascun flag di convergenza. I protocolli di comunicazioni utilizzati sono:

- **AXI LITE** per il trasferimento dei parametri e i valori di ritorno con un basso *bitwidth* come it_{max} e B_c.
- **MASTER AXI** per il trasferimento di grandi quantità di dati come B_R, B_P, B_u e $B_{u'}$.

Il protocollo **AXI LITE** è stato implementato utilizzando il blocco *AXI Interconect* che collega due porte master del PS a due porte slave del kernel FPGA. Il protocollo **MASTER AXI** è stato invece implementato utilizzando il blocco *AXI SmartConnect* che collega una porta slave del master a 6 porte master del kernel FPGA.

3.4 Worst Case Speed-Up

Uno dei parametri incogniti del kernel è la dimensione del batch B_{size}. Si vuole quindi studiare come varia lo *speed-up* ottenuto dall'utilizzo del kernel FPGA al variare di B_{size} in modo da scegliere il valore ottimale.
Sia N il numero di patch da elaborare, allora saranno necessarie

$$N_B = \left\lceil \frac{N}{B_{size}} \right\rceil$$

invocazioni al kernel *BatchAlign2D*. Sia t_{patch} il tempo massimo impiegato dal kernel FPGA per completare un batch di patch. Se l'ottimizzazione su ogni patch del batch è effettuata in parallelo allora t_{patch} non dipende da B_{size}. Per questo t_{patch} può essere interpretato come il tempo necessario ad ottimizzare una patch che non raggiunge la convergenza entro it_{max} iterazioni del metodo *Gauss-Newton*. Ponendo t_{prep} pari al tempo necessario a preparare le *region* di un batch e $t_{param}(B_{size})$ il tempo necessario a trasferire i dati di ogni batch tra PS e FPGA, si ha che il tempo totale massimo necessario per elaborare N patch in batch da B_{size} è

$$T^{max}_{FPGA}(N, B_{size}) = N_B \cdot (t_{patch} + t_{param}(B_{size})) + t_{prep}$$

dove solamente il t_{prep} del primo batch influisce sul tempo totale a causa dell'*interleaving*.
Vogliamo quindi scegliere un valore di B_{size} che garantisca di avare uno *speed-up* minimo

$$S^{min}(B_{size}) = \frac{T_{PS}(N)}{T^{max}_{FPGA}(N, B_{size})} > 1 \qquad \forall N > 0$$

Per fare questo possiamo limitarci a trovare il valore di B_{size} che permette di avere *speed-up* nel *worst-case*. Il caso peggiore si ha creando un *work-unbalance* per il kernel, ovvero per ogni batch tutte le patch raggiungono la convergenza in una sola iterazione, tranne una che non riesce a convergere in it_{max} iterazioni. Si ha quindi che il tempo di esecuzione del kernel è sempre pari a $T^{max}_{FPGA}(N, B_{size})$ per tutti gli N_B batch, mentre il carico di lavoro del PS è ridotto al minimo. In questa configurazione possiamo modellare il tempo impiegato dal PS come

$$T_{PS}(N) = (N - N_B) \cdot t_{min} + N_B \cdot t_{max}$$

dove t_{min} è il tempo richiesto dal PS per eseguire l'inizializzazione e una iterazione del metodo *Align2D*, mentre t_{max} è il tempo richiesto dal PS per eseguire l'inizializzazione e it_{max} iterazioni del metodo *Align2D*. Allo stesso modo possiamo calcolare lo *speed-up* nel *best-case*, ovvero quando il carico di lavoro è massimo. Questo lo si ha quando tutte le patch richiedono it_{max} o più iterazioni per convergere nella fase di ottimizzazione *Gauss-Newton*.
Si ha quindi che

$$S^{max}(B_{size}) = \frac{T^{max}_{PS}(N)}{T^{max}_{FPGA}(N, B_{size})} > 1 \qquad \forall N > 0$$

con

$$T_{PS}^{max}(N) = N \cdot t_{max}$$

Misurando t_{patch}, $t_{param}(B_{size})$, t_{prep}, t_{min} e t_{max} possiamo quindi avere un'idea dello *speed-up* ottenuto nel *worst-case* e nel *best-case*. Si hanno così due funzioni che rappresentano un *lower-bound* e un *upper-bound* dello *speed-up* che si può ottenere per diversi valori di N in casi di lavoro più generici.

3.5 Risultati

Un benchmark per testare il kernel è stato ricavato estraendo quasi un migliaio di patch eseguendo la versione originale dell'algoritmo su dei video di testing per VO. Nella Tabella 3 sono stati riportati i tempi misurati su questo benchmark mentre in Figura 8 è riportato lo *speed-up* nel *worst-case* e nel *best-case* calcolati utilizzando il modello prima descritto e i tempi raccolti. Dal grafico possiamo quindi vedere che se il numero di patch da elaborare è sufficientemente alto, il kernel FPGA riesce comunque a fornire uno *speed-up* superiore a 1 anche nel *worst-case*. I casi in cui non vi è *speed-up* sono quelli in cui il numero di patch da elaborare è minore di B_{size} o quando il numero di patch di elaborare fa si che l'ultimo batch non sia completamente pieno. Invece, nel caso in cui il lavoro è massimo si ottiene uno *speed-up* che oscilla tra 1.2 e 2.8, ma che si assesta tra 2 e 2.8 già per un numero di patch superiore a 50.

$\mathbf{B_{size}}$	$\mathbf{t_{min}}$	$\mathbf{t_{max}}$	$\mathbf{t_{prep}}$	$\mathbf{t_{param}(B_{size}) + t_{patch}}$
16	0.062702583	0.143403504	0.226008	0.845937

Tabella 3: Tempi di esecuzione in ms.

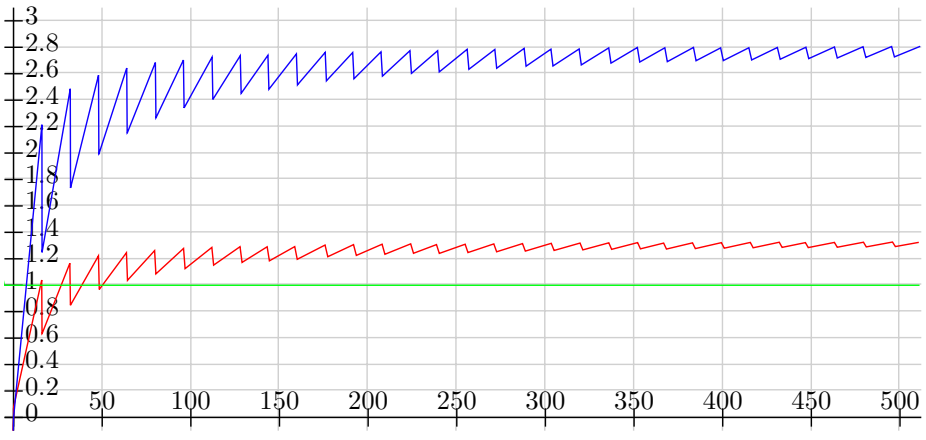

Figura 8: Speed-up nel *worst-case* e nel *best-case* al variare del numero di patch da processare con B_{size} pari a 16.

4 Conclusioni

In conclusione, si ritiene che l'utilizzo dell'FPGA possa portare vantaggi sopratutto nel caso particolare in cui ci siano numerosi feature points. A parte questo, il lavoro svolto alla base dell'elaborato ha cercato di seguire una metodica nella risoluzione di problemi tramite l'FPGA:

1. Comprensione ed analisi del problema
2. Profiling
3. Identificazione dei colli di bottiglia
4. Identificazione dell'algoritmo da ottimizzare
5. Preparazione di test per lo *speed-up*
6. Implementazione dell'algoritmo da ottimizzare su PS come base-line
7. Studio degli aspetti di ottimizzazione di FPGA
8. Progettazione del modulo da implementare su FPGA e progettazione di un algoritmo di controllo che venga incontro all'FPGA (le tre idee: batch di feature points, regioni di immagini, double buffering a livello di batch)
9. Implementazione del modulo FPGA
10. Implementazione dell'algoritmo di controllo
11. Testing e misurazioni

Una cosa che è emersa è che la programmazione dell'FPGA uno strumento potente, ma molto time-consuming. Per questo gli algoritmi implementati sono stati pensati e testati prima su CPU e preceduti da una valutazione sulle caratteristiche (punti di forza e punti di deboelzza) dell'hardware, in quanto tentare direttamente sul hardware è molto oneroso. Legato a questo, ritorna infatti utile una formula che dia idea dei parametri di configurazione ottimali per ottenere le migliori performance.
Sebbene si sia ottenuto uno *speed-up* positivo, non si ritiene non sia possibile ottenere performance migliori, in particolare è probabilmente possibile fare più leva sulla parallelizzazione al crescere delle dimensioni del batch, tuttavia si ritiene di aver raggiunto un buon risultato, sopratutto considerando la natura piuttosto sequenziale del carico di lavoro ottimizzato.

Riferimenti bibliografici

[1] Ros, 2016.

[2] Oprofile, 2019.

[3] Simon Baker and Iain Matthews. Lucas-kanade 20 years on: A unifying framework. *International Journal of Computer Vision*, 56(3):221 – 255, March 2004.

[4] Michael Burri, Janosch Nikolic, Pascal Gohl, Thomas Schneider, Joern Rehder, Sammy Omari, Markus W Achtelik, and Roland Siegwart. The euroc micro aerial vehicle datasets. *The International Journal of Robotics Research*, 2016.

[5] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. SVO: Fast semi-direct monocular visual odometry. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2014.

[6] Christian Forster, Matia Pizzoli, and Davide Scaramuzza. Svo with ros, 2016.

[7] Davide Scaramuzza and Friedrich Fraundorfer. Tutorial: visual odometry. *IEEE Robotics and Automation Magazine*, 18(4):80–92, 2011.

[8] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry: Part ii: Matching, robustness, optimization, and applications. *IEEE robotics & automation magazine*, 18(4):80–92, 2011.

[9] Wikipedia. Metodo dei cofattori — wikipedia, l'enciclopedia libera, 2011.