Università degli Studi
di Modena e Reggio Emilia

DIPARTIMENTO DI SCIENZE FISICHE, INFORMATICHE E MATEMATICHE
Corso di Laurea in Informatica

Accelerazione di algoritmi di Stereo Matching su GPU per sistemi SLAM

Relatore:
Dott. Filippo Muzzini
Correlatore:
Prof. Nicola Capodieci

Tesi di Laurea di:
Luca Anzaldi

Anno Accademico 2023/2024

Indice

Capitolo 1

Introduzione

L'evoluzione delle tecnologie informatiche ha portato a una crescente necessità di eseguire calcoli complessi in tempi ridotti, specialmente nei settori che richiedono grandi potenze computazionali come la grafica, l'intelligenza artificiale e la simulazione scientifica. Per far fronte a queste esigenze, si sono sviluppati linguaggi e framework specifici per la programmazione parallela, che permettono di sfruttare le capacità di elaborazione simultanea di più core o unità di calcolo.

Questa tesi si propone di analizzare e implementare un percorso di ottimizzazione per un algoritmo di **Stereo Matching** utilizzando la tecnologia **CUDA** (Compute Unified Device Architecture), sviluppata da NVIDIA per l'elaborazione parallela su GPU. Il problema del Stereo Matching è una delle questioni fondamentali nel campo della visione artificiale e consiste nel calcolare la disparità tra due immagini catturate da punti di vista leggermente diversi (come accade negli occhi umani) al fine di ricostruire la profondità e ottenere una rappresentazione tridimensionale della scena.

Gli algoritmi di **Stereo Matching** tipicamente richiedono un'intensa capacità di calcolo, soprattutto quando si cerca di ottenere risultati accurati su immagini ad alta risoluzione. L'obiettivo principale di questa tesi è ottimizzare l'esecuzione di uno di questi algoritmi sfruttando le caratteristiche dell'elaborazione parallela offerta dalle GPU tramite CUDA. La scelta di utilizzare **CUDA** è motivata dalla sua capacità di distribuire il carico computazionale su un numero elevato di core, rendendo possibile l'elaborazione simultanea di grandi quantità di dati, e migliorando significativamente le prestazioni rispetto all'implementazione su **CPU tradizionali**.

Nella trattazione verrà esaminato l'approccio usato per ottimizare l'algoritmo, con un'attenzione particolare all'efficienza e alla precisione, e si approfondiranno i principi di ottimizzazione parallela, analizzando tecniche come la decomposizione del problema in thread e la gestione ottimale della memoria della **GPU**. La tesi illustrerà anche i vantaggi e le sfide nell'implementare un algoritmo di Stereo Matching su architettura parallela, discutendo sia gli aspetti teorici che quelli pratici, con test sperimentali che metteranno a confronto le performance dell'algoritmo ottimizzato rispetto a versioni non parallele o meno efficienti.

Capitolo 2

Panoramica dei linguaggi paralleli e specifiche di CUDA

2.1 Introduzione

La programmazione parallela si basa sull'esecuzione contemporanea di più istruzioni o blocchi di codice, sfruttando risorse hardware multiprocessore, multicore o acceleratori come le GPU (**Graphics Processing Units**). A differenza della programmazione sequenziale tradizionale, che esegue le istruzioni una dopo l'altra, la programmazione parallela permette di dividere i problemi in sottoproblemi più piccoli, i quali possono essere risolti simultaneamente, aumentando l'efficienza complessiva del sistema.

Uno dei linguaggi di programmazione parallela più diffusi è **CUDA** (Compute Unified Device Architecture), un framework sviluppato da *NVIDIA*. Esso consente di sfruttare la potenza delle GPU per eseguire calcoli ad alte prestazioni, trasformando le unità grafiche in potenti strumenti di elaborazione general-purpose. Attraverso CUDA, gli sviluppatori possono scrivere codice in **C**, **C++** o Python, che viene successivamente parallelizzato e distribuito tra le varie unità di calcolo della GPU.

Inoltre, esistono altri linguaggi e API orientati alla programmazione parallela, come OpenCL, che offre un approccio più generalista e multi-piattaforma, permettendo di sfruttare anche altre tipologie di hardware, come **CPU** e **FPGA**. Questi strumenti sono oggi fondamentali per affrontare i carichi di lavoro che richiedono un'enorme quantità di calcoli in tempi contenuti, portando significativi benefici in campi come il machine learning, la crittografia, l'elaborazione video e le simulazioni fisiche.

2.2 Differenza di architettura tra CPU e GPU

L'architettura della CPU e della GPU può essere confrontata in maniera sintetica. La CPU è progettata per ridurre la latenza, ossia cerca di ottenere i risultati delle operazioni nel minor tempo possibile. Per fare questo, dispone di **cache L1** di grandi dimensioni, che aiutano a ridurre la latenza media di accesso ai dati, e utilizza poche unità logiche aritmetiche ad alte prestazioni per calcolare velocemente i risultati. I modelli moderni di CPU sfruttano anche il parallelismo a livello di istruzione, elaborando in anticipo risultati parziali per ridurre ulteriormente i tempi di attesa. Al contrario, l'architettura della GPU è orientata al **throughput**, ovvero al volume di operazioni elaborate simultaneamente. Poiché contiene un numero elevato di processori paralleli, non può dotarli di cache L1 grandi come quelle delle CPU. Di conseguenza, gli accessi alla memoria sono più lenti, causando maggiori latenze. Tuttavia, quando la GPU esegue molti più thread rispetto ai suoi core fisici (situazione chiamata "*over-subscription*"), riesce a nascondere queste latenze passando rapidamente l'esecuzione da un thread all'altro.

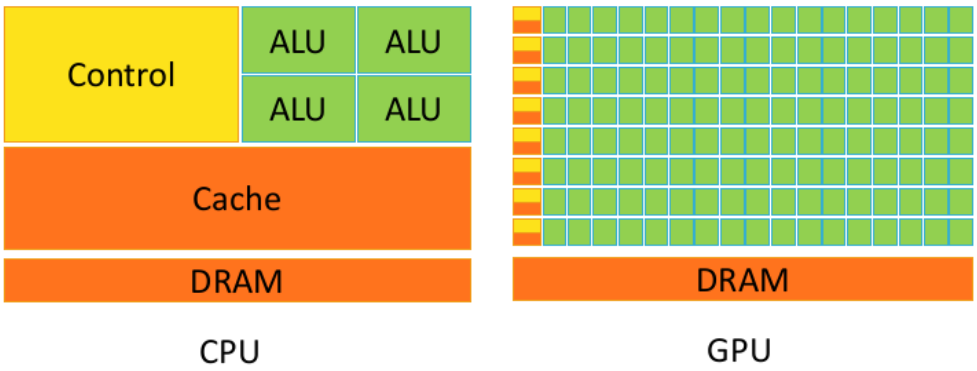

Figura 2.1: Differenza di architettura [Kerbl et al., 2022]

I **thread** della GPU sono molto più leggeri rispetto a quelli della CPU, il che rende più efficiente il loro passaggio da uno all'altro. Anche se le latenze possono essere più elevate, la capacità di commutare rapidamente i thread e di gestire più istruzioni in parallelo permette alla GPU di mantenere un **throughput elevato** durante l'elaborazione. Pertanto, i vantaggi di utilizzare le GPU per calcoli intensivi aumentano all'aumentare del numero di thread impiegati per un determinato compito.

2.3 Il modello di esecuzione di CUDA

Nel modello di esecuzione di CUDA esistono due tipi di funzioni principali:

- `__global__`: Queste funzioni sono chiamate *kernel functions* e possono essere invocate dalla *host* (CPU) per essere eseguite sulla *device* (GPU). Sono definite con il qualificatore `__global__` e devono essere invocate con una sintassi speciale, specificando il numero di thread e blocchi da lanciare. Una caratteristica particolare delle funzioni `__global__` è che il loro tipo di ritorno deve essere sempre `void`. Ecco un esempio di dichiarazione di una funzione `__global__`:

```
1    __global__ void myKernel(int *data) {
2        // Codice da eseguire sulla GPU
3    }
```

- `__device__`: Queste funzioni possono essere chiamate solo da altre funzioni eseguite sulla *device* (ovvero dalla GPU stessa) e non possono essere invocate direttamente dalla *host*. Le funzioni `__device__` sono eseguite sulla GPU e possono avere qualsiasi tipo di ritorno. Ecco un esempio di dichiarazione di una funzione `__device__`:

```
1    __device__ int square(int x) {
2        return x * x;
3    }
```

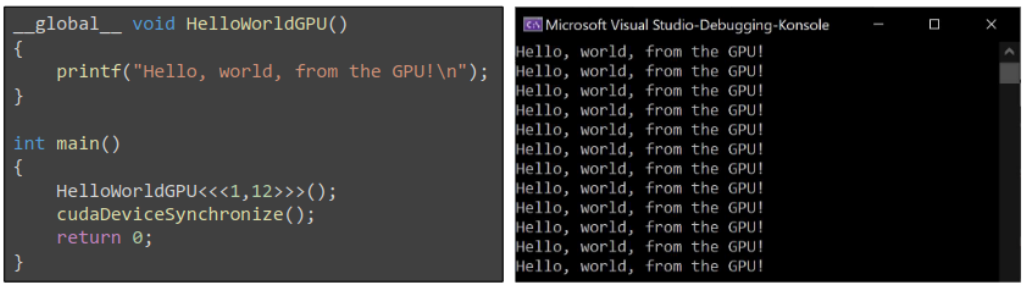

Figura 2.2: Kernel di base (1 blocco, 12 thread) [Kerbl et al., 2022]

2.4 Gerarchia di esecuzione di CUDA

La gerarchia di esecuzione in CUDA si basa su un'architettura parallela organizzata in livelli, che consente di suddividere il lavoro complessivo in unità

più piccole. Il codice viene eseguito in parallelo su molte unità di calcolo, con la possibilità di gestire grandi quantità di operazioni simultaneamente. Questa struttura permette di scalare l'esecuzione in base alle esigenze, ottimizzando l'uso delle risorse disponibili per migliorare le prestazioni complessive.

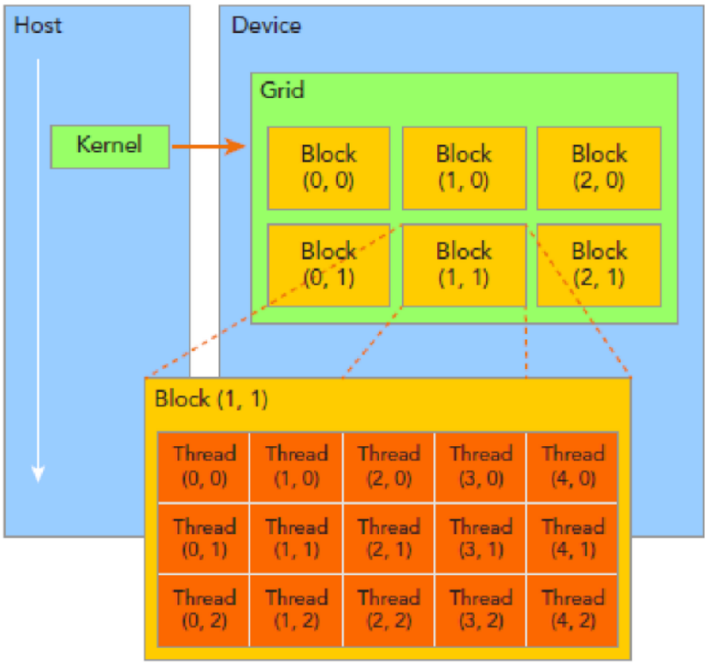

Figura 2.3: Illustrazione della gerarchia di esecuzione di CUDA [Miaoyu Cai, 2022]

Thread

Il *thread* è l'unità di base di esecuzione in CUDA. Ogni thread esegue un'istanza del codice definito in un *kernel*. I thread possono essere identificati tramite un identificatore univoco (`threadIdx`) che li distingue dagli altri all'interno del loro *block*.

Figura 2.4: Thread nel modello di CUDA [Gupta, 2020]

Block

I thread sono organizzati in *block*, che rappresentano un gruppo di thread che può essere eseguito insieme sulla GPU. Ogni block è indipendente dagli altri, e ogni thread all'interno di un block è identificato da un indice (`threadIdx`). La dimensione massima di un block dipende dall'architettura della GPU, ma tipicamente contiene fino a 1024 thread. L'indice del block all'interno della griglia è fornito da `blockIdx`.

Figura 2.5: Blocco(block) nel modello di CUDA [Gupta, 2020]

Grid

I block sono organizzati in una *grid*. Una grid è una collezione di block, e la dimensione della grid è determinata al momento del lancio del kernel. L'indice del block nella grid è indicato da `blockIdx`, che permette ai block di essere distinti e indirizzati individualmente.

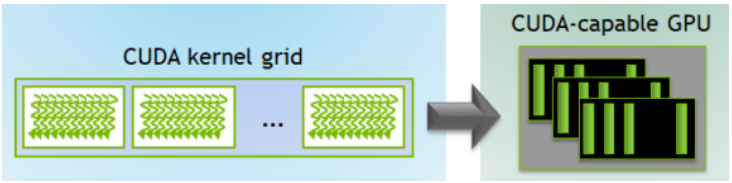

Figura 2.6: Griglia(grid) nel modello di CUDA [Gupta, 2020]

Warp

Un concetto fondamentale nella gerarchia di esecuzione di CUDA è il *warp*. Un warp è un insieme di **32 thread** che vengono eseguiti in parallelo all'interno di un singolo block. La GPU esegue i thread in gruppi di warp, e tutti i thread di un warp seguono lo stesso percorso di esecuzione (flusso di controllo), il che significa che i thread all'interno di un warp sono soggetti a esecuzione sincrona. Se i thread di un warp prendono percorsi diversi (ad esempio in caso di branch divergenti), la GPU eseguirà i percorsi separatamente, riducendo l'efficienza.

2.5 Sincronizzazione

Di base, CUDA presume che, se una funzione può dipendere da un'altra, tale dipendenza sarà garantita. Pertanto, CUDA gestisce la sincronizzazione tra le diverse funzioni per assicurare che vengano eseguite nell'ordine corretto. Ecco alcuni esempi:

- Il kernel B è successivo al kernel A. Allora il kernel B aspetterà che il primo kernel abbia finito l'esecuzione.
- Se stiamo scrivendo i dati sulla GPU il driver aspetterà ad eseguire la funzione mettendola in coda.
- Se vogliamo scrivere i dati dalla GPU → CPU il driver aspetterà che la funzione in esecuzione nella GPU sia terminata.

È possibile ovviamente sincronizzare manualmente l'esecuzione in un programma CUDA, in modo tale da controllare l'ordine di esecuzione tra kernel, thread o tra operazioni tra CPU e GPU. I principali metodi di sincronizzazione sono i seguenti:

- **Sincronizzazione tra Host e Device:**
  - `cudaDeviceSynchronize()`: Questa funzione viene chiamata dalla CPU per aspettare che tutte le operazioni precedenti lanciate sulla GPU siano completate. Viene utilizzata per assicurarsi che il kernel abbia terminato l'esecuzione prima di continuare con altre operazioni sul codice host.
- **Sincronizzazione tra Thread di un Kernel:**
  - `__syncthreads()`: Questa funzione viene chiamata all'interno di un kernel e sincronizza tutti i thread all'interno di un blocco. Ogni thread deve raggiungere il punto in cui è presente `__syncthreads()` prima di poter proseguire, ed è utile quando c'è bisogno che tutti i thread completino un'operazione comune prima di procedere ulteriormente. Va notato che `__syncthreads()` funziona solo all'interno di un singolo blocco di thread, non tra diversi blocchi.
- **Stream e Eventi per Sincronizzazione Personalizzata:**
  - CUDA supporta l'uso di stream e eventi per una sincronizzazione più avanzata. Gli stream permettono di lanciare kernel in modo

asincrono e indipendente tra loro, mentre gli eventi vengono utilizzati per sincronizzare l'esecuzione tra diversi stream o per misurare il tempo di esecuzione tra operazioni.

2.6 Tipi di memoria

Nell'architettura CUDA, esistono diversi tipi di memoria, ognuno con caratteristiche specifiche in termini di accesso, latenza e visibilità.

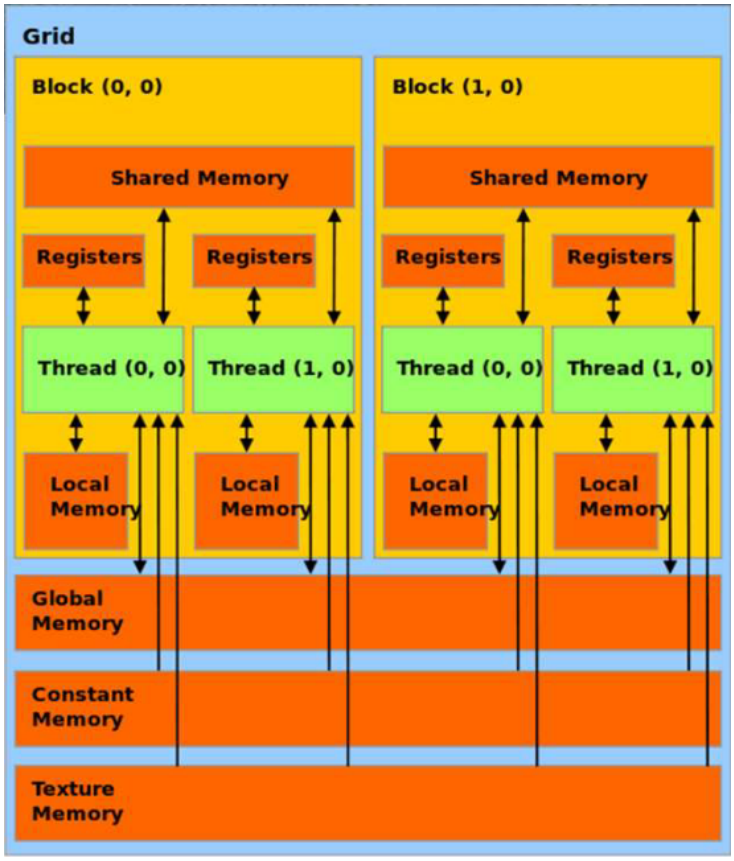

Figura 2.7: Tipi di memoria [Kerbl et al., 2022]

2.6.1 Memoria condivisa (Shared Memory)

La *memoria condivisa* è una memoria veloce e a bassa latenza, accessibile da tutti i thread all'interno dello stesso blocco. È utile per condividere dati tra thread e ridurre l'accesso alla memoria globale, ottimizzando le prestazioni

dei programmi paralleli. La memoria condivisa è limitata in quantità e deve essere gestita con attenzione.

2.6.2 Memoria globale (Global Memory)

La *memoria globale* è la memoria principale accessibile da tutti i thread e blocchi. Essa ha una latenza elevata, quindi è preferibile minimizzarne l'utilizzo, soprattutto se si può sfruttare la memoria condivisa. Tuttavia, la memoria globale ha una capacità molto maggiore rispetto alla memoria condivisa.

2.6.3 Memoria locale (Local Memory)

La *memoria locale* è assegnata ai singoli thread e viene utilizzata per memorizzare variabili private. Nonostante il nome, questa memoria è fisicamente una parte della memoria globale, e pertanto ha tempi di accesso relativamente lenti.

2.6.4 Memoria costante (Constant Memory)

La *memoria costante* è una memoria a sola lettura che viene utilizzata per valori che rimangono invariati durante l'esecuzione di un kernel. Essendo memorizzata nella cache, ha una latenza più bassa rispetto alla memoria globale quando viene letta simultaneamente da più thread.

2.6.5 Memoria di texture (Texture Memory)

La *memoria di texture* è progettata per il recupero efficiente di dati strutturati e può essere utilizzata per migliorare le prestazioni in applicazioni che richiedono accessi irregolari ai dati. Viene spesso usata in ambiti grafici, ma è accessibile anche per scopi generali.

2.6.6 Registri

I *registri* sono la forma di memoria più veloce in CUDA e sono utilizzati per memorizzare variabili temporanee all'interno dei thread. Ogni thread ha accesso ai propri registri, che sono molto limitati in numero. Un uso inefficiente dei registri può portare al cosiddetto *spill* nella memoria locale, con conseguente rallentamento delle prestazioni.

Capitolo 3

Panoramica della localizzazione e mappatura simultanea (SLAM)

3.1 Introduzione al SLAM

La Localizzazione e Mappatura Simultanea, comunemente nota come **SLAM** (*Simultaneous Localization and Mapping*), rappresenta uno dei pilastri fondamentali della robotica moderna e della visione artificiale. SLAM si riferisce al processo computazionale mediante il quale un agente autonomo, come un robot o un veicolo, costruisce una mappa dell'ambiente circostante mentre contemporaneamente determina la propria posizione all'interno di essa.

3.1.1 Importanza e applicazioni del SLAM

L'importanza del SLAM risiede nella sua capacità di conferire autonomia a sistemi robotici in ambienti non strutturati o sconosciuti. Le applicazioni del SLAM sono molteplici e in rapida espansione:

1. **Robotica mobile**: SLAM è essenziale per la navigazione autonoma di robot in ambienti interni ed esterni.
2. **Veicoli autonomi**: Automobili e droni self-driving utilizzano tecniche SLAM per navigare in sicurezza.

3. **Realtà aumentata (AR)**: SLAM permette il posizionamento accurato di oggetti virtuali nel mondo reale.
4. **Esplorazione spaziale**: Robot su altri pianeti utilizzano SLAM per navigare e mappare terreni sconosciuti.
5. **Ispezioni industriali**: SLAM facilita l'automazione di ispezioni in ambienti complessi come impianti industriali.
6. **Servizi di emergenza**: Può assistere nelle operazioni di ricerca e salvataggio in aree disastrate.

3.1.2 Sfide principali nel campo del SLAM

1. **Incertezza e rumore**: I sensori non sono perfetti e introducono errori nelle misurazioni.
2. **Ambienti dinamici**: Oggetti in movimento possono complicare la costruzione di mappe coerenti.
3. **Loop closure**: Riconoscere luoghi già visitati è cruciale ma computazionalmente costoso.
4. **Scalabilità**: Mantenere performance in tempo reale su mappe di grandi dimensioni.
5. **Robustezza**: Garantire un funzionamento affidabile in condizioni ambientali variabili.
6. **Efficienza computazionale**: Ottimizzare l'utilizzo delle risorse, specialmente su piattaforme con capacità limitate.

3.1.3 Introduzione ai sistemi ORB-SLAM

I sistemi **ORB-SLAM** (Oriented FAST and Rotated BRIEF Simultaneous Localization and Mapping) rappresentano uno stato dell'arte nell'ambito della localizzazione e mappatura simultanea *(SLAM)* in robotica e visione artificiale. Questi sistemi si distinguono per la loro capacità di costruire mappe tridimensionali dell'ambiente circostante in tempo reale, utilizzando unicamente input visivi originati da telecamere monoculari, stereo o RGB-D.

Il processo completo può essere sintetizzato in questo modo:

1. Acquisizione delle immagini
2. Rilevamento dei punti caratteristici ORB

3. Tracciamento dei punti caratteristici
4. Costruzione della mappa
5. Localizzazione
6. Ottimizzazione della mappa e rivelamento dei loop.

3.1.4 Caratteristiche principali di ORB-SLAM

Rilevamento delle caratteristiche (features)

Per ogni immagine acquisita, l'algoritmo ORB estrae punti di riferimento unici chiamati **caratteristiche** (features) utilizzando un algoritmo chiamato **FAST** (Features from Accelerated Segment Test). Questi punti caratteristici possono essere angoli o dettagli unici nell'immagine, come il bordo di una finestra o l'angolo di un edificio. Questi punti devono essere facilmente riconoscibili anche da diverse angolazioni o distanze.

Figura 3.1: Ottenimento dei punti chiave in un frame [Ganti and Waslander, 2018]

Tracciamento dei punti caratteristici

Quando la telecamera si muove, l'algoritmo confronta i punti caratteristici delle nuove immagini con quelli delle immagini precedenti. Questo permette di capire come si è spostata la telecamera: se un punto si è spostato a destra nell'immagine, potrebbe significare che la telecamera si è mossa a sinistra, e viceversa. Questo è possibile grazie ai descrittori binari **BRIEF** (Binary Robust Independent Elementary Features) che tengono salvati le features principali.

Localizzazione e Mappatura

Durante il movimento della telecamera e l'acquisizione sequenziale di immagini, il sistema ORB-SLAM sfrutta i punti caratteristici estratti per generare e aggiornare incrementalmente una rappresentazione tridimensionale dell'ambiente circostante(**mappatura**).

Simultaneamente al processo di mappatura, il sistema esegue una stima continua della posa della telecamera, comprendente posizione e orientamento, in relazione alla mappa tridimensionale in fase di costruzione. Questo processo di localizzazione consente al dispositivo di determinare con precisione le sue coordinate spaziali all'interno dell'ambiente in fase di ricostruzione (**localizzazione**).

Ottimizzazione della mappa e rivelamento dei loop.

Il sistema gestisce una mappa di punti chiave e riconosce i ritorni (**loop closures**) per correggere eventuali errori accumulati nella stima della posizione, migliorando così l'accuratezza della mappa e della localizzazione.

Riassunto - Tracking, Mappatura e Rilevamento di Loop

ORB-SLAM divide il processo in tre thread principali: **tracking** (tracciamento della fotocamera), **mappatura** (costruzione e aggiornamento della mappa) e **rilevamento di loop** (riconoscimento di posizioni precedentemente visitate) con successiva chiusura del loop.

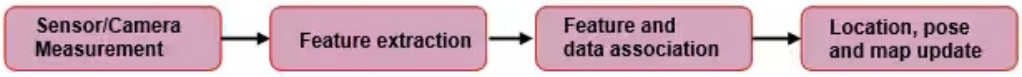

Figura 3.2: Diagramma a blocchi generico e semplificato del processo SLAM basato sulle caratteristiche [Babu, 2024]

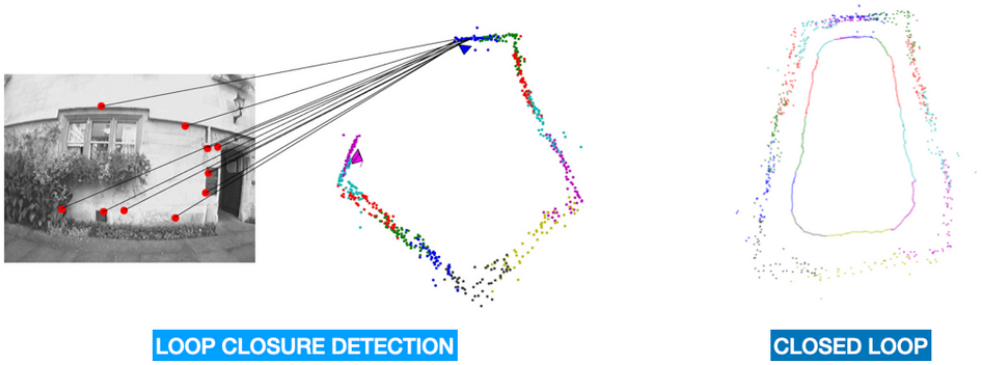

Figura 3.3: Rappresentazione algoritmo di loop closure [Unknown(Blog), 2023]

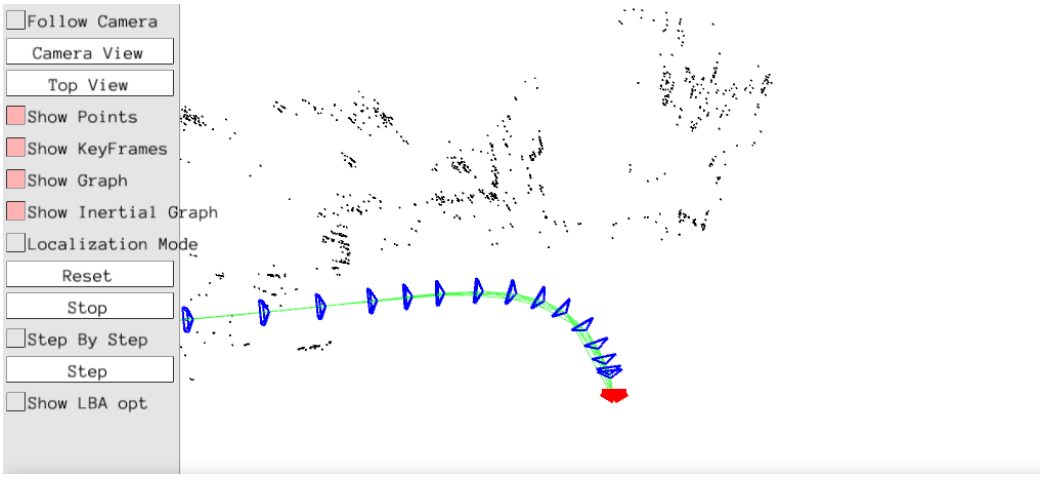

Figura 3.4: "Cattura" dello schermo durante la generazione della mappa ambientale

Capitolo 4

Analisi e implementazione delle ottimizzazioni

4.1 Introduzione

L'obiettivo di questo capitolo è ottimizzare un algoritmo di **Stereo Matching** attraverso la parallelizzazione. In particolare, l'algoritmo contiene già una fase di estrazione *ORB (Oriented FAST and Rotated BRIEF)* parallelizzata, ma vi sono ancora altre parti del processo che possono essere migliorate sfruttando la parallelizzazione.

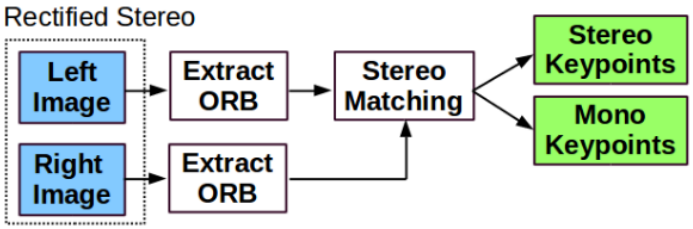

Figura 4.1: Stato iniziale del programma [Mur-Artal and Tard´os, 2017]

Per raggiungere questo obiettivo, il linguaggio di programmazione CUDA verrà utilizzato per implementare soluzioni che consentano di sfruttare le capacità di calcolo parallelo delle GPU, al fine di ridurre il tempo di elaborazione dell'algoritmo e migliorare le prestazioni complessive. Nelle seguenti sezioni verrano mostrati i passaggi per la realizzazione del progetto.

4.2 Analisi delle possibili funzioni da ottimizzare

Durante l'analisi del codice di ORB-SLAM3, è stata individuata la funzione

```
1 void Frame::ComputeStereoMatches()}
```

che si occupa dell'esecuzione della parte di **Stereo Matching**. Successivamente è stata eseguita una schematizzazione ed è stata analizzatata la sua struttura per identificare le possibili aree su cui intervenire per effettuare le ottimizzazioni.

4.2.1 Schema generale - ComputeStereoMatches()

Lo schema generale del funzionamento della funzione `ComputeStereoMatches()` è illustrato nell'immagine sottostante. Tale schema si compone di tre macrofunzioni principali e mostra il numero medio di iterazioni eseguite, permettendo di vedere il carico computazionale svolto.

1. **Raccolta dei keypoints:** La prima funzione si occupa di selezionare, per ogni punto chiave(keypoint) dell'immagine destra, un intorno e di aggiungerlo a una struttura dati apposita, preparandolo per le fasi successive.
2. **Selezione del miglior candidato:** La seconda funzione si concentra sulla ricerca del miglior candidato tra quelli selezionati nella fase precedente, valutando i vari intorni e scegliendo quello più adatto.
3. **Filtraggio dei risultati:** La terza funzione applica filtri ai keypoints migliori, al fine di mantenere solo quelli che soddisfano determinati criteri di qualità, migliorando così la precisione complessiva del processo.

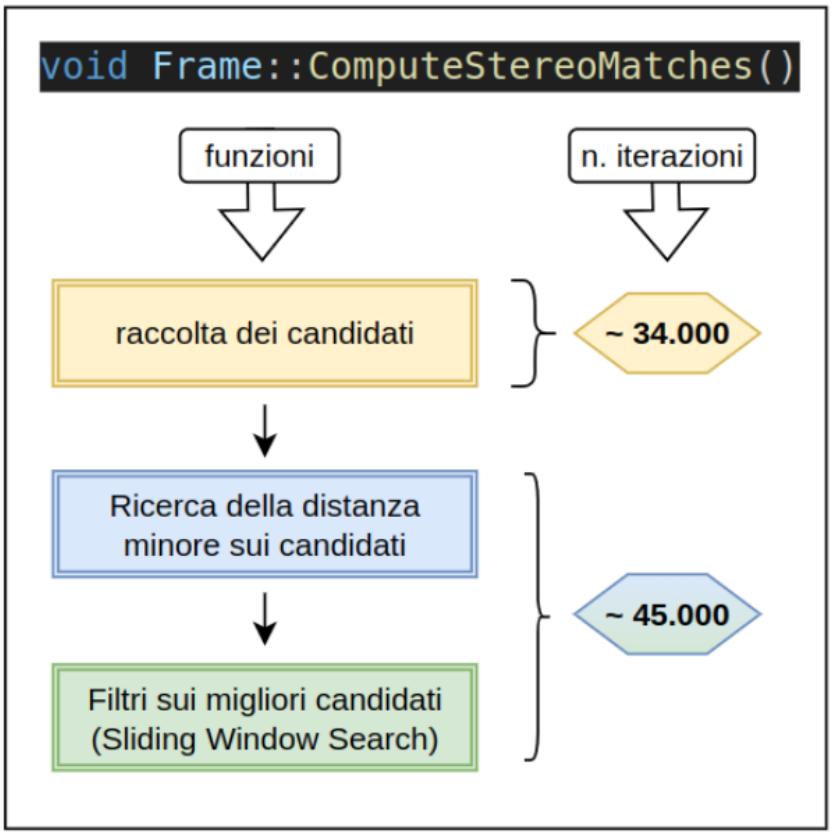

Figura 4.2: Schematizzazione della funzione *ComputeStereoMatches()*

4.2.2 Schema della ricerca delle distanze minime

Nell'illustrazione sottostante è possibile visionare la schematizzazione della funzione addetta a trovare i migliori punti chiave tra quelli candidati (cioè quelli con distanza minore dall'immagine a sinistra)

Ci sono due tipi di rappresentazioni. Quella ad **alta livello** che è astratta e non tiene conto delle strutture dati. Mentre la seconda si concentra sulle **strutture dati utilizzate** e **le iterazioni** eseguite per raggiungere il risultato atteso.

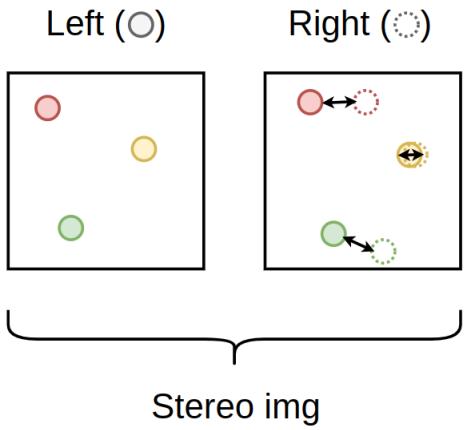

Figura 4.3: Schematizzazione ad alto livello (CPU)

Come si può vedere nell'immagine questa parte dell'algoritmo calcolerà la varie distanze, tra i punti **candidati trovati** ed i punti effettivi nell'**immagine sinistra**. E successivamente andrà a scartare i candidati non ottimali.

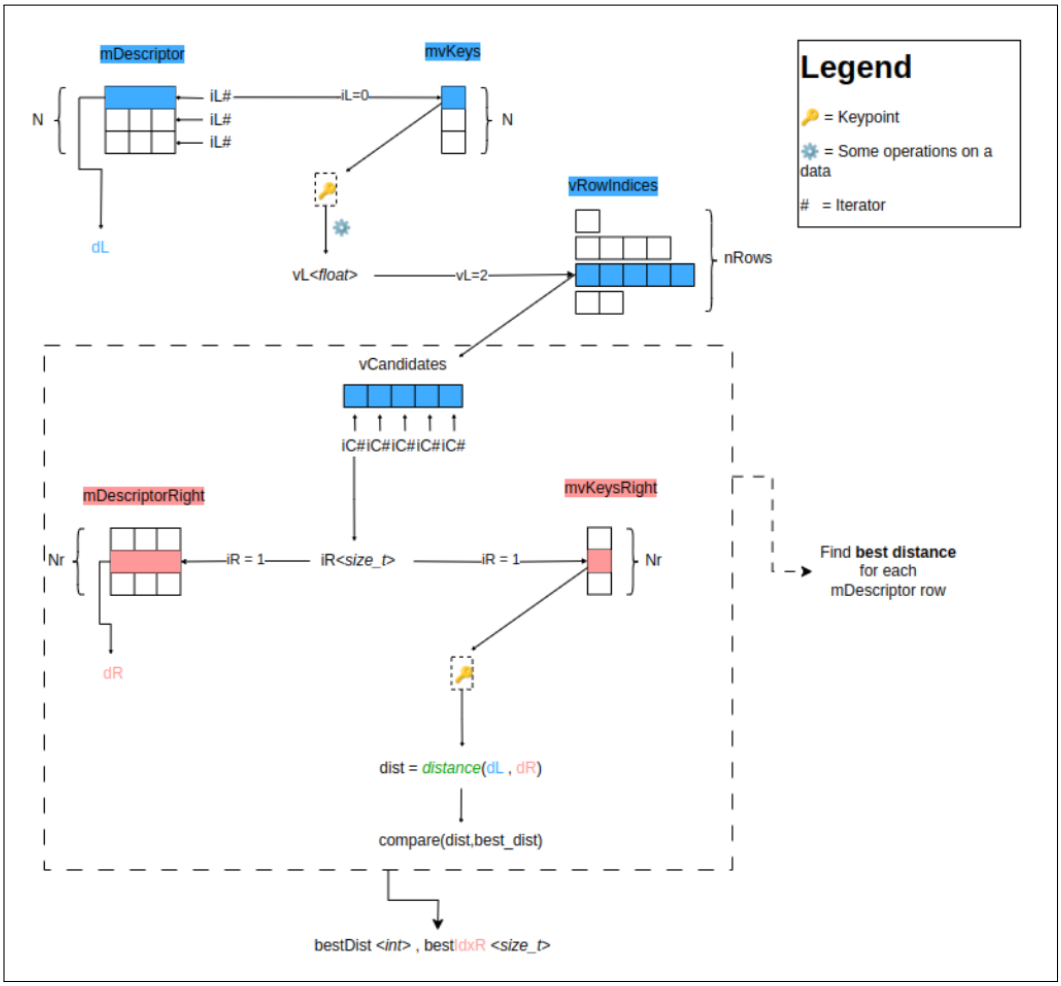

Figura 4.4: Schematizzazione a basso livello (CPU)

4.2.3 Schema del filtraggio finale sui punti chiave

In questa sezione, analizzeremo il processo di Sliding Window nel contesto del nostro sistema ORB-SLAM 3 nella versiona non ottimizata per CPU.

1. L'immagine seguente illustra schematicamente la componente **Sliding Window** del sistema:

2. È importante notare che il filtraggio associato alla Sliding Window viene eseguito successivamente al calcolo della distanza minore di un singolo punto chiave. Questo permette, eventualmente, di non attendere il calcolo di tutte le distanze ma di eseguire il filtraggio ad ogni singolo punto chiave.

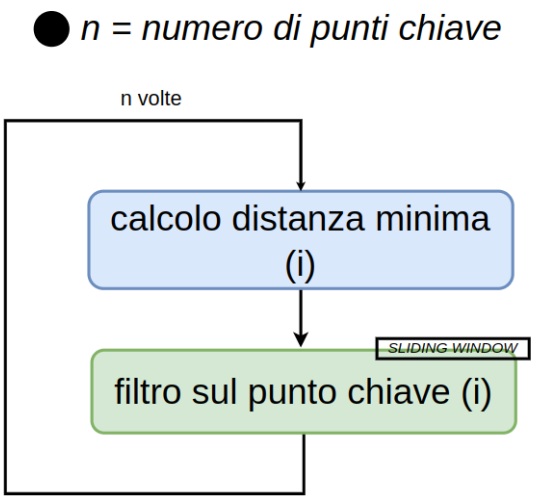

Figura 4.5: Schematizzazione a alto livello (CPU)

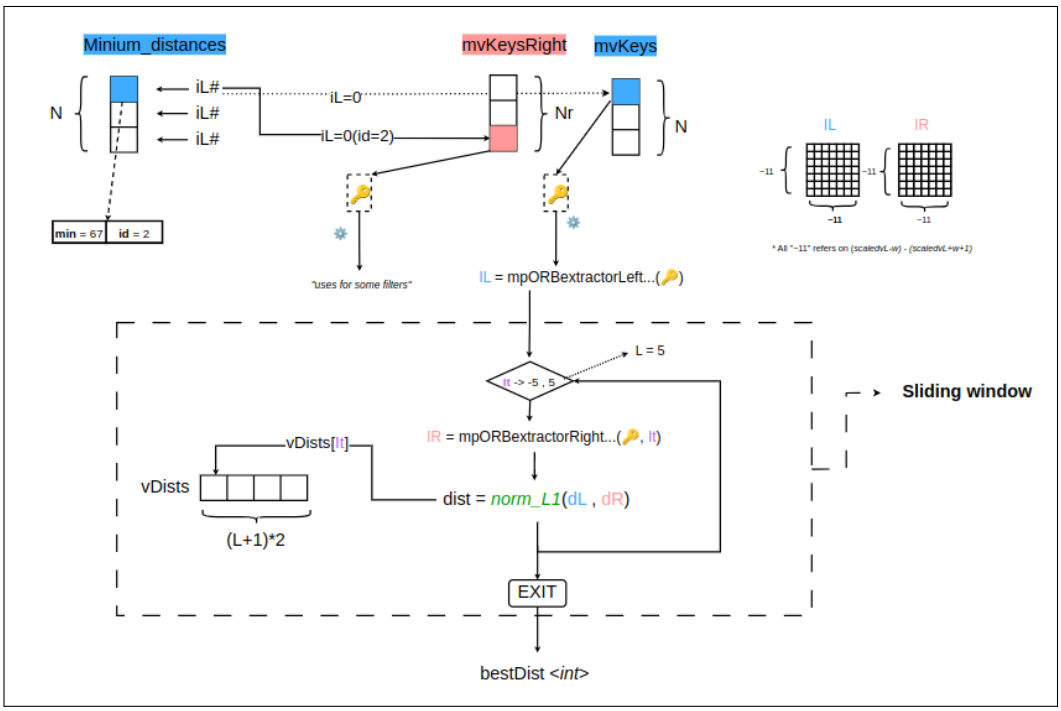

Figura 4.6: Schematizzazione a basso livello (CPU)

4.3 Ottimizzazione della funzione "ComputeStereoMatches()"

In questa sezione, verranno illustrati dettagliatamente i passaggi necessari per l'ottimizzazione della funzione **computeStereoMatches()** al fine di sfruttare appieno le capacità di elaborazione parallela offerte dalle GPU. L'obiettivo principale di questa ottimizzazione è migliorare significativamente le prestazioni dell'algoritmo di **stereo matching**, riducendo i tempi di esecuzione e aumentando l'efficienza complessiva del sistema. Verranno esaminati i seguenti aspetti chiave del processo di ottimizzazione:

1. Il ragionamento alla base dell'ottimizzazione
2. La realizzazione di un programma intermedio per avvicinarci alla soluzione finale.
3. La distribuzione delle sotto-funzioni nei diversi dispositivi (*GPU* oppure *CPU*).
4. Lo schema specifico per la "**ricerca delle distanze minime**" (4.2.2)
5. Lo schema specifico per il "**filtraggio sui punti chiave**" (4.2.3)

Questa esplorazione ci permetterà di valutare l'efficacia dell'approccio adottato e di trarre conclusioni sulle potenzialità dell'ottimizzazione in questo contesto specifico.

4.3.1 Percorso logico verso l'ottimizzazione

Il processo di sviluppo della soluzione finale si è articolato attraverso diverse fasi metodologiche:

1. Si è proceduto all'identificazione della funzione di **Stereo Matching** all'interno del codice sorgente di ORB-SLAM.
2. L'attenzione è stata focalizzata sui costrutti iterativi **for**, i quali, in un contesto di esecuzione *single-thread*, possono risultare particolarmente onerosi dal punto di vista computazionale.
3. è stata effettuata un'analisi per determinare il **numero medio di iterazioni** di tali costrutti. I risultati di queste analisi sono stati riportati nello schema generale (si rimanda al capitolo [4.2.1]).
4. In conclusione, si è proceduto ad un esame approfondito delle implementazioni delle funzioni preesistenti, seguito dalla progettazione di versioni

parallele mediante l'utilizzo di *CUDA*, con l'obiettivo di ottenere una significativa riduzione dei tempi di elaborazione.

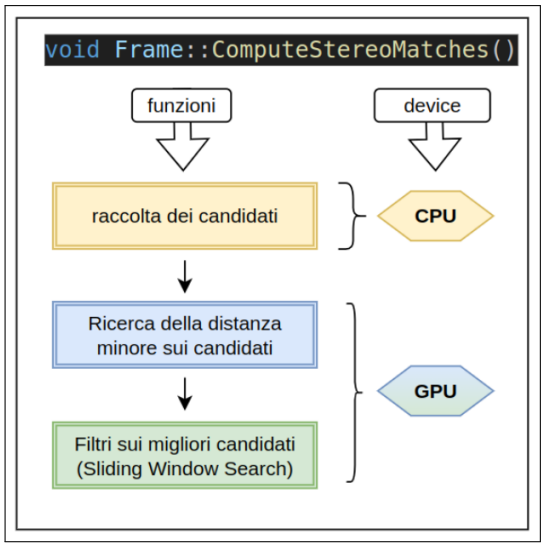

Figura 4.7: Progettazione della distribuzione delle funzioni su device

4.3.2 Avvicinamento alla soluzione

Nel percorso verso la realizzazione del nostro obiettivo finale, si è ritenuto opportuno sviluppare un programma intermedio. Questo approccio ha permesso di affrontare gradualmente la complessità del problema, fornendo preziosi spunti e una base solida su cui costruire.

È importante sottolineare che questo programma intermedio, pur condividendo alcune similitudini con il nostro obiettivo finale, presenta delle **differenze significative** poichè non tiene conto di alcuni aspetti fondamentali:

- L'incremento di strutture dati necessarie
- L'adattamento della struttura dei dati da $CPU \rightarrow GPU$
- Dei numerosi filtri di controllo eseguiti, durante le iterazioni dei punti chiave.

Gli **elementi comuni** che si possono trovare invece sono:

- L'utilizzo della stessa struttura dati *Vcandidates[]* che rappresenta una matrice avente il numero di colonne non omogeneo.

- L'esecuzione di operazioni sugli elementi della matrice (di complessità notevolmente inferiore al programma finale).
- La ricerca di un punto minimo nella struttura dati principale.

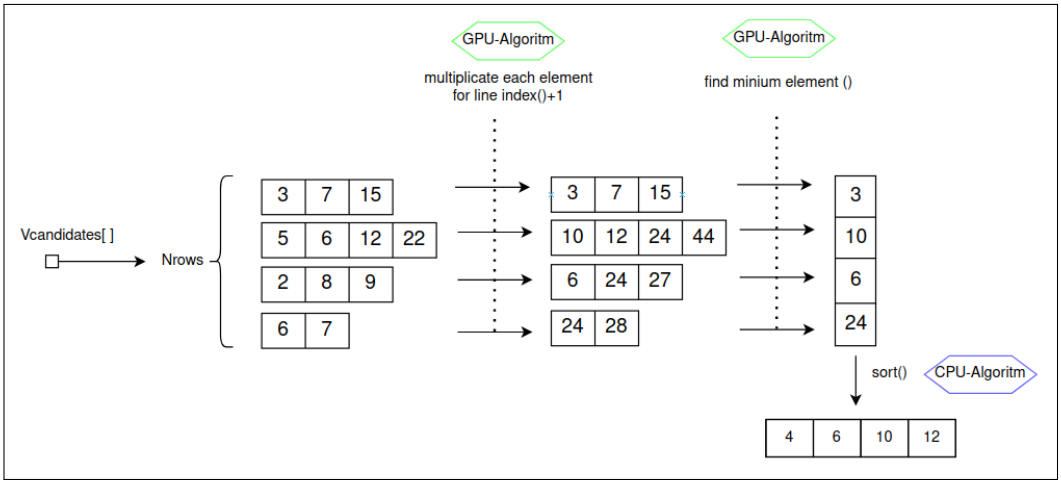

Figura 4.8: Schemattizazione del programma di "avvicinamento"

4.3.3 Parallelizzazione della "ricerca delle distanze minime"

Per implementare la parallelizzazione della funzione di ricerca delle distanze minime, abbiamo seguito un approccio strutturato che ha richiesto la creazione di nuovi file e la modifica di quelli esistenti. Di seguito, descriviamo i passaggi principali di questo processo:

Creazione di nuovi file

Il primo passo è stato la creazione di due nuovi file:

- `gpu_stereoMatches.cu`: Questo file conterrà il codice CUDA per la parallelizzazione della funzione.
- `gpu_stereoMatches.h`: Il file header corrispondente, che conterrà le dichiarazioni delle funzioni e le strutture dati necessarie.

In questi file, abbiamo implementato la logica per la parallelizzazione della ricerca delle distanze minime, sfruttando le capacità di elaborazione parallela offerte dalle GPU.

Aggiornamento del CMakeLists.txt

Per assicurarci che i nuovi file fossero inclusi correttamente nel processo di compilazione, abbiamo aggiunto le seguenti righe al file `CMakeLists.txt`:

```
src/gpu_stereoMatches.cu
include/gpu_stereoMatches.h
```

Questa modifica garantisce che il compilatore riconosca e includa i nostri nuovi file CUDA durante il processo di build.

Inclusione dell'header nel file Frame.cc

Per rendere le nuove funzionalità accessibili all'interno del nostro framework esistente, abbiamo aggiunto la seguente linea di inclusione nel file `Frame.cc`:

```
#include "gpu_stereoMatches.h"
```

Questo passaggio assicura che le funzioni e le strutture dati definite nei nostri nuovi file siano disponibili per l'uso all'interno del contesto del frame.

Progettazione della struttura del kernel

Dopo aver preparato l'infrastruttura necessaria, ci siamo concentrati sulla progettazione della struttura del kernel CUDA da lanciare. Questa fase ha comportato:

- L'analisi dell'algoritmo di ricerca delle distanze minime per identificare le parti parallelizzabili.
- La definizione della griglia e dei blocchi di thread per ottimizzare l'esecuzione sulla GPU.
- La progettazione delle funzioni del kernel per sfruttare efficacemente l'architettura parallela della GPU.

La struttura del kernel è stata progettata per massimizzare il parallelismo, permettendo l'elaborazione simultanea di molteplici punti di interesse e riducendo significativamente il tempo di esecuzione complessivo dell'algoritmo di ricerca delle distanze minime.

Questo è stato possibile grazie all'implementazione di una singola funzione, garantendo così un'uniformità nell'utilizzo delle risorse computazionali in termini di thread, blocchi e griglie. Questo approccio ha consentito una gestione efficiente e coerente delle unità di elaborazione parallela. A seguito si procederà ad una analisi più dettagliata dell'implementazione.

Calcolo della distanza

Nell'immagine sottostante è possibile notare che usando lo stesso numero di blocchi e thread, andiamo ad eseguire operazioni su due differenti strutture dati diverse:

- mDescriptor (matrice)
- vRowIndices (matrice con colonne eterogenee)

È importante notare che il numero 370 si riferisce al numero di righe della struttura dati *vRowIndices*. Questo dato potrebbe cambiare in futuro se si modifica la risoluzione delle fotocamere.

Inoltre si nota che per adattare la dimensione della matrice *mDescriptor* si è deciso di utilizzare il *partition_factor*. Questa variabile permette di raggruppare più righe nello stesso blocco e quindi evitare di lanciare blocchi "a vuoto".

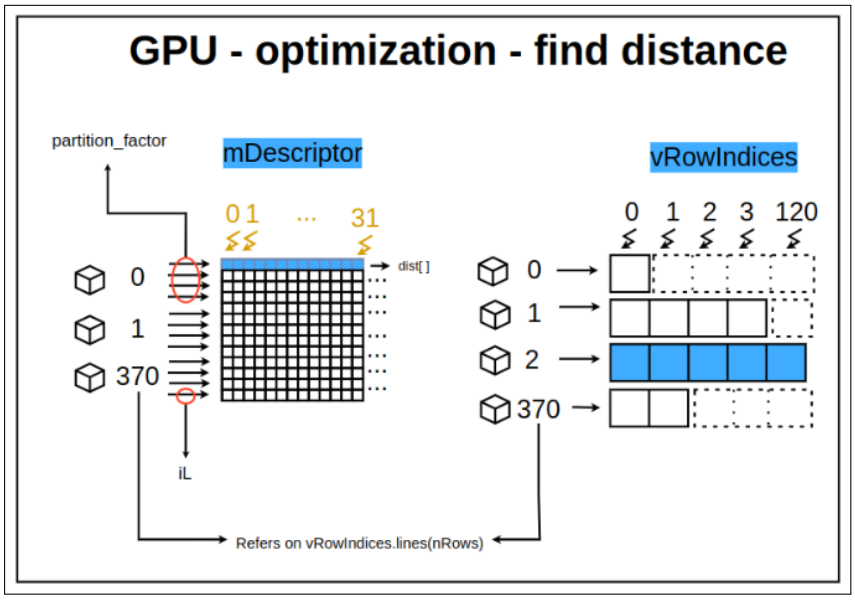

Figura 4.9: Configurazione della griglia (calcolo distanza)

Calcolo della distanza minima

Una volta calcolate le distanze, vengono confrontate tra loro e inserite, in modo opportuno, in un array la cui dimensione è determinata dal *partition_factor*. È importante notare che viene creato un vettore distinto per ciascun blocco istanziato.

Come passaggio finale tutti i 370 vettori composti da *partition_factor* elementi, verranno spostati in un vettore di dimensione pari a N (N = numero di righe della matrice *mDescriptor*)

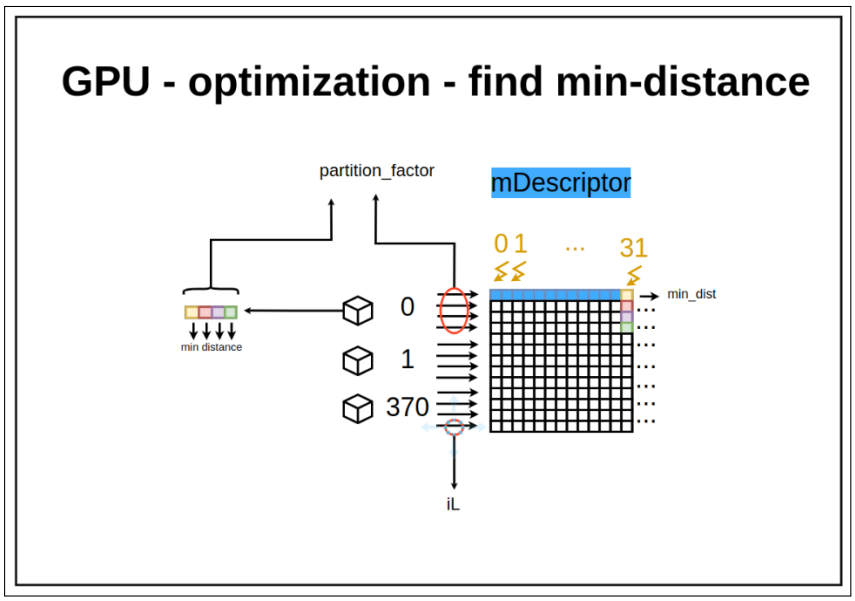

Figura 4.10: Configurazione della griglia (calcolo distanza minima)

Riscrittura funzione sequenziale (Distanza di Hamming)

Abbiamo modificato la funzione che calcola la distanza di Hamming in modo che potesse essere chiamata direttamente dalla GPU. Questo è stato realizzato convertendo la funzione in una funzione di tipo **__device__**. Tale modifica permette alla funzione di essere eseguita direttamente sul dispositivo GPU, ottimizzando così le prestazioni e riducendo la necessità di trasferimenti di dati tra CPU e GPU.

CODE : Algoritmo per calcolare distanza di Hamming

```
__device__ int DescriptorDistance(const unsigned
   char *a, const unsigned char* b){

    int dist=0;

    const int32_t* a_int = reinterpret_cast<const
       int32_t*>(a);
    const int32_t* b_int = reinterpret_cast<const
       int32_t*>(b);

    for(int i=0; i<8; i++) {
        unsigned int v = a_int[i] ^ b_int[i];
        v = v - ((v >> 1) & 0x55555555);
        v = (v & 0x33333333) + ((v >> 2) & 0
           x33333333);
        dist += (((v + (v >> 4)) & 0xF0F0F0F) * 0
           x1010101) >> 24;
    }

    return dist;
}
```

Confronto precisione risultati

```
{1} Percentuale somiglianza valori : 100.00%
{1} Percentuale somiglianza indici :  97.33%
```

Figura 4.11: Risultato accuratezza dell'algoritmo

4.3.4 Parallelizzazione del "filtraggio delle distanze"

In questa parte dell'esperimento, abbiamo utilizzato i file configurazione del programma citati nella sezione precedente [4.3.3]. Questo approccio ci ha permesso di mantenere una coerenza metodologica e di costruire sulla base delle conoscenze già acquisite. Tuttavia, la differenza sostanziale in questa fase è stata l'introduzione di una nuova funzione di tipo **kernel()**, specifica per l'implementazione CUDA.

Configurazione del programma

1. `gpu_stereoMatches.cu`: Contenente il codice CUDA per la parallelizzazione.
2. `gpu_stereoMatches.h`: Il file header corrispondente.

Progettazione della struttura del kernel

A seguito è stata progettata la **configurazione della griglia** in modo tale da sfruttare a pieno il parallelismo e minimizzare l'utilizzo di registri della GPU.

In questa funzione parallela CUDA si è deciso di lanciare K *blocchi* e J *thread* (K = N/256 , J = 256). È opportuno ricordare che N fa riferimento al numero di righe della struttura dati *mDescriptor* in cui sono stati salvati le distanze minime (approfondimento nella sezione [4.3.3])

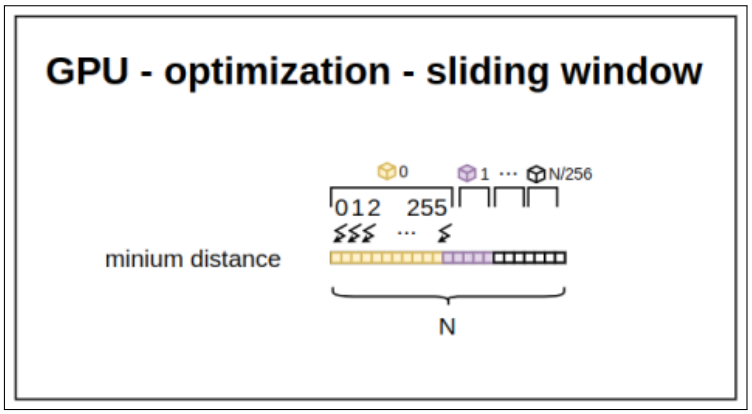

Figura 4.12: Configurazione della griglia(sliding window)

Scrittura metodi *getter()* per l'immagine piramidale

Visionando la schematizzazione a basso livello (CPU) presente nel capitolo [4.2.3] si può notare che vengono estratte due sottomatrici *IL* e *IR*. Queste sottomatrici vengono ottenute a seguito di questa riga di codice:

```
1 cv::Mat IL = mpORBextractorLeft->mvImagePyramid[kpL.
    octave].rowRange(scaledvL-w,scaledvL+w+1).
    colRange(scaleduL-w,scaleduL+w+1);
```

Il problema principale è che *mvImagePyramid* è definita nel seguente modo:

```
std::vector<cv::Mat> mvImagePyramid;
```

Come si può osservare, ci troviamo di fronte a un *vector* di tipo *cv::Mat*. Tuttavia, è importante notare che queste due strutture dati non sono compatibili con le funzioni CUDA, le quali supportano esclusivamente array standard.

La soluzione a questo problema è stata quella di definire all'interno del file **ORBSLAM3/include/ORBextractor.h** un metodo **getter()** che permettesse di ottenere la struttura dati primitiva definita così :

```
uchar *d_images;      //liv (0[EMPTY],1,2,3,4,5,6,7)
uchar *d_inputImage; //immagine originale (liv 0)
```

La struttura primitiva si riferisce ad un **Octave** cioè una serie di immagini della stessa scena ma con risoluzioni diverse, ottenute riducendo progressivamente la dimensione dell'immagine originale. Ogni octave contiene più versioni filtrate a varie scale, usate per rilevare caratteristiche invarianti alla scala, come nei metodi SIFT o ORB.

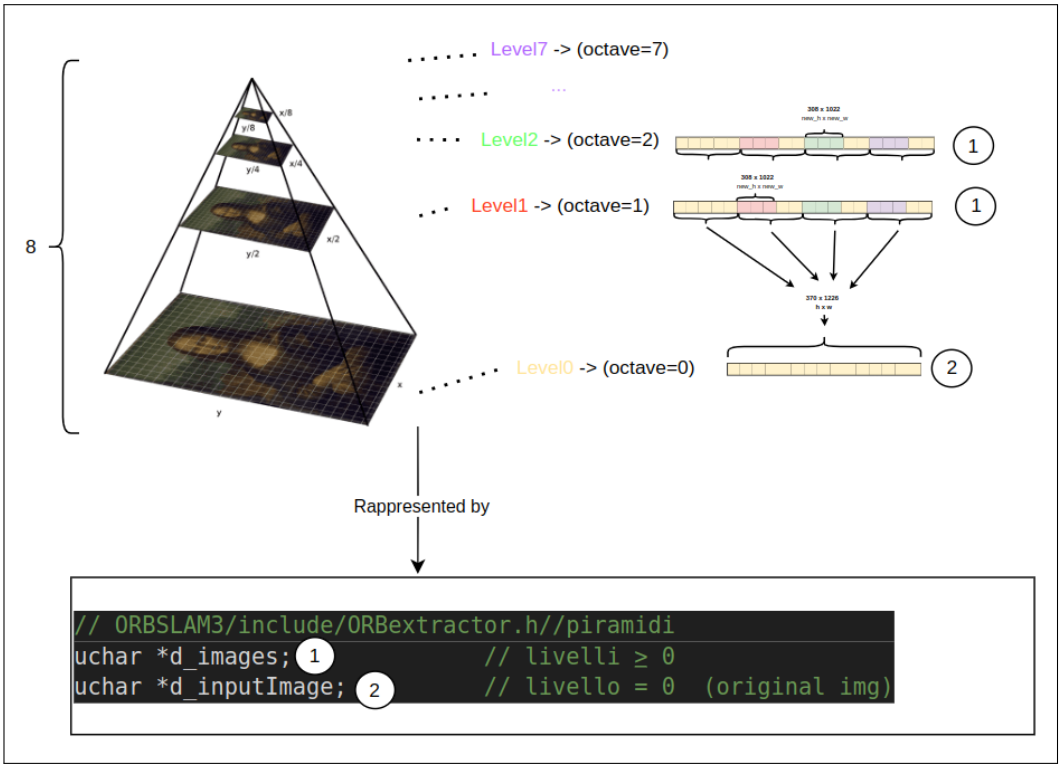

Figura 4.13: Rappresentazione visiva dell'octave

Riscrittura funzione sequenziale (Norma Manhattan)

Abbiamo riscritto la funzione che calcola la norma 1, nota anche come "distanza di Manhattan". Poichè questa funzione, che nel programma CPU originale era stata implementata utilizzando la libreria OpenCV non poteva essere lanciata nella funzione parallela.

Per questo motivo è stata riscritta come una funzione di tipo **__device__**.

```
__device__ float norm1(const uchar *V1 , const uchar * V2 , int size , int i1 , int j1 , int i2 , int j2 , int cols1 , int cols2 , int incR , int iL){

    float sum = 0;
    int countRow = 0;
    int countCol = 0;
    int j1_temp = j1;
    int j2_temp = j2;
    while(countRow < size){
        while(countCol < size){
            int index1 = ( (i1*cols1) + j1 );
            int index2 = ( (i2*cols2) + j2 );
            sum = sum + abs(((float) V1[index1] - ( float) V2[index2]));
            countCol++;
            j1++;
            j2++;
        }
        i1++;
        i2++;
        j1 = j1_temp;
        j2 = j2_temp;
        countCol = 0;
        countRow++;
    }

    return sum;

}
```

È importante ricordare la definizione di norma 1 e il suo scopo: la norma 1, anche nota come distanza di Manhattan o distanza cityblock, è definita come "*la somma dei valori assoluti delle differenze delle coordinate*". Nel nostro contesto, viene utilizzata per misurare la dissimilarità tra due vettori di caratteristiche, fornendo una metrica robusta per confrontare descrittori di immagini e identificare corrispondenze tra punti di interesse.

$$\|v\|_1 = \sum_{i=1}^{n} |v_i|$$

Figura 4.14: Definizione della Norma di Manhattan

Confronto precisione risultati

```
{2} mathing pairs vDistIdx(CPU vs GPU) : 96.666664 %

{2} mathing pairs mvuRight(CPU vs GPU) : 99.950638 %

{2} mathing pairs mvDepth (CPU vs GPU) : 99.950638 %
```

Figura 4.15: Risultato accuratezza dell'algoritmo

Capitolo 5

Esperimenti e risultati

5.1 Introduzione agli esperimenti

Nel contesto della nostra ricerca si è pensato di progettare una serie di esperimenti per valutare l'efficacia delle nostre modifiche e per trovare eventuali variazioni utili a sfruttare l'ottimizzazione realizzata. Questa sezione delinea la metodologia sperimentale, gli strumenti utilizzati e l'ambiente hardware in cui sono stati condotti gli esperimenti.

5.1.1 Valutazione della Qualità

La qualità degli esperimenti sarà valutata utilizzando un programma specializzato scritto in Python. Questo software è stato sviluppato specificamente per l'analisi e la valutazione di sistemi di odometria visiva e SLAM, con particolare attenzione alle prestazioni su dataset KITTI.

Il programma di valutazione è disponibile pubblicamente e può essere accessibile al seguente link: https://github.com/Huangying-Zhan/kitti-odom-eval

5.1.2 Ambiente Hardware e Software

Gli esperimenti sono stati condotti su due diversi dispositivi. Il primo dispositivo è un **laptop** con le con le seguenti specifiche tecniche:

- CPU: 12th Gen Intel(R) Core(TM) i7-1280P
- GPU: GeForce RTX 3060 Mobile / Max-Q , 6Gb
- RAM: 16 GB DDR4-3200

- Storage: 1 TB NVMe SSD
- Sistema Operativo: Linux mint 21.3
- CUDA Driver Version: 535.183.01
- CUDA Version: 12.2

Il secondo dispositivo è una **board** detta anche **modulo**. Il sito ufficiale di *NVIDIA* definisce che "*la gamma di moduli Jetson Xavier™ include i primi computer del mondo progettati appositamente per le macchine autonome, con prestazioni IA fino a 32 TOPS che li rendono ideali per gestire l'odometria grafica, la fusione di sensori, la localizzazione e la mappatura, il rilevamento di ostacoli e gli algoritmi di pianificazione dei percorsi fondamentali per i robot di nuova generazione*" [NVIDIA-website, 2023b]

La specifiche tecniche della board utilizzata sono quelle evidenziate in arancione e si riferiscono al modello **Nvidia Jetson Xavier AGX 32GB**:

Specifiche tecniche

	Jetson AGX serie Xavier		
	AGX Xavier 64 GB	AGX Xavier	AGX Xavier Industrial
Prestazioni IA	32 TOPS		30 TOPS
GPU	Architettura NVIDIA Volta con 512 NVIDIA CUDA Core e 64 Tensor Core		
CPU	CPU NVIDIA Carmel ARM v8.2 8 core a 64-bit 8 MB L2 + 4 MB L3		
Acceleratore DL	2 NVDLA		
Acceleratore visivo	2 PVA		
Engine cluster di sicurezza	-		2 Arm Cortex-R5 in lockstep
Memoria	LPDDR4x 64 GB 256-bit 136,5 GB/s	LPDDR4x 32 GB 256-bit 136,5 GB/s	LPDDR4x 32 GB 256-bit (supporto ECC) 136,5 GB/s

Figura 5.1: Nvidia Jetson Xavier AGX 32GB [NVIDIA-website, 2023a]

5.1.3 Procedura Sperimentale

Per eseguire gli esperimenti e la valutazione, seguire il seguente tutorial script:

1. Spostarsi nella directory `/kitti-odom-eval`.
2. Eseguire il comando: `conda env update --file <nome_file.yml> --prune`
3. Attivare l'ambiente conda con: `conda activate kitti_eval`
4. Spostare il file "CameraTrajectory.txt" nella directory `/result` e rinominarlo in "`<SEQ>.txt`" (Esempio: "07.txt").
5. Eseguire il comando di valutazione:

```
python3 eval_odom.py --result result --seqs <sequence> --align scale
```

Questa procedura ci permetterà di valutare in modo consistente e riproducibile le prestazioni del nostro sistema ottimizzato, confrontandole con i risultati di riferimento del dataset KITTI.

Nei paragrafi seguenti, presenteremo i risultati ottenuti da questi esperimenti e discuteremo le implicazioni per il nostro lavoro di ottimizzazione.

5.2 Esperimento 1 - Variazione raggio dei candidati su CPU

In questa fase del nostro studio, procederemo a modificare il parametro **radiusMultiplier** della funzione che raccoglie i **candidati** per poi eseguire la funzione **ComputeStereoMatches** (4.2.1) su CPU.

L'obiettivo è analizzare l'impatto di queste variazioni sulle prestazioni complessive del sistema. Nello specifico, valuteremo i potenziali vantaggi e svantaggi derivanti da ciascuna modifica, sia in termini di **implicazioni temporali** e in termini di **precisione**.

Qualora si riscontrino significativi aumenti nei tempi di elaborazione, considereremo l'opportunità di **ottimizzare il processo** mediante l'utilizzo di GPU. L'implementazione di soluzioni di calcolo parallelo su GPU potrebbe consentirci di mitigare gli effetti negativi osservati, sfruttando la capacità di elaborazione massiccia di queste unità hardware.

Il fine ultimo di questa analisi è individuare una configurazione ottimale che bilanci efficacemente la qualità dei risultati e l'efficienza computazionale. Attraverso questa metodologia di sperimentazione e ottimizzazione, miriamo a sviluppare **una versione migliorata del sistema**, che rappresenti un avanzamento significativo rispetto all'implementazione originale.

CODE : Codice relativo all'inserimento dei candidati

```
for(int iR=0; iR<Nr; iR++){
    const cv::KeyPoint &kp = mvKeysRight[iR];
    const float &kpY = kp.pt.y;
    // Variabile modifica per esperimento
    const float radiusMultiplier = 2.0
    const float r = radiusMultiplier *mvScaleFactors
       [mvKeysRight[iR].octave];
    const int maxr = ceil(kpY+r);
    const int minr = floor(kpY-r);
    // Inserimento candidati nell'intorno del raggio
    for(int yi=minr;yi<=maxr;yi++){
        vRowIndices[yi].push_back(iR);
        size_refer[yi]++;
    }
}
```

5.2.1 Analisi dell'Impatto del parametro Radius: Risultati e Limiti

L'esperimento condotto per valutare l'influenza del parametro 'radius' sulle prestazioni del sistema ORB-SLAM 3 ha portato a risultati che possiamo considerare positivi, soprattutto in termini di *miglioramento della precisione*.

Le variazioni del valore del raggio in aumento, hanno prodotto dei miglioramenti termini di precisione per il valore ATE(m).

- *Kitti04 / Kitti07* : Miglioramento precisione ATE per un moltiplicatore di raggio pari a **16**.
- *Kitti06* : Miglioramento precisione ATE per un moltiplicatore di raggio pari a **4**.

l'**ATE** (Absolute Trajectory Error) è una misura utilizzata per valutare la precisione della traiettoria stimata dal sistema di SLAM rispetto alla traiettoria di riferimento (ground truth).

Si osserva inoltre che i tempi nella parte di **Stereo Matching** mostrano un aumento significativo, con incrementi che arrivano fino al 100%. Si può prevedere che questo problema diventi trascurabile con la parallelizzazione dell'algoritmo. L'aumento dei tempi di elaborazione, infatti, potrà essere mitigato sfruttando l'elaborazione parallela, consentendo di distribuire il carico computazionale e ridurre significativamente i tempi complessivi.

Tabella 5.1: Esperimenti variazione raggio (KITTY DATASET - 04) [laptop]

Radius multiplier	1.0	2.0	4.0	8.0	16.0
Translation error	0.47	0.46	0.46	0.47	0.47
Rotational error	0.07	0.08	0.09	0.07	0.07
ATE (m)	0.73	0.77	0.75	0.78	0.71
RPE (m)	0.017	0.017	0.017	0.017	0.017
RPE (deg)	0.031	0.032	0.031	0.030	0.030
ORB Extraction (ms)	9.1419	8.9207	8.9707	9.0029	8.9305
Stereo Matching (ms)	3.0216	3.391	3.962	4.8351	6.345
ORB Extraction (σ)	61.54	57.737	58.214	58.98	58.148
Stereo Matching (σ)	0.54303	0.63088	0.68821	0.74869	0.6854

Tabella 5.2: Esperimenti variazione raggio (KITTY DATASET - 06) [laptop]

Radius multiplier	1.0	2.0	4.0	8.0	16.0
Translation error	0.48	0.48	0.46	0.48	0.48
Rotational error	0.15	0.16	0.15	0.16	0.15
ATE (m)	2.43	2.45	2.29	2.59	2.31
RPE (m)	0.013	0.013	0.014	0.014	0.014
RPE (deg)	0.031	0.031	0.032	0.032	0.032
ORB Extraction (ms)	6.0595	6.0632	6.0573	6.0988	6.1996
Stereo Matching (ms)	2.832	3.231	3.732	4.678	6.281
ORB Extraction (σ)	28.606	28.451	28.551	28.839	28.839
Stereo Matching (σ)	0.534	0.557	0.645	0.716	0.681

Tabella 5.3: Esperimenti variazione raggio (KITTY DATASET - 07) [laptop]

Radius multiplier	1.0	2.0	4.0	8.0	16.0
Translation error	0.48	0.44	0.47	0.46	0.47
Rotational error	0.24	0.23	0.25	0.26	0.26
ATE (m)	0.52	0.49	0.5	0.54	0.46
RPE (m)	0.014	0.013	0.014	0.014	0.014
RPE (deg)	0.04	0.039	0.041	0.040	0.040
ORB Extraction (ms)	6.305	6.3717	6.356	6.280	6.679
Stereo Matching (ms)	3.087	3.573	4.072	4.896	6.679
ORB Extraction (σ)	29.31	28.58	29.45	28.65	29.87
Stereo Matching (σ)	0.535	0.526	0.642	0.689	0.688

Nella sezione successiva analizzeremo i risultati ottenuti in termini di tempi di esecuzione a seguito della parallelizzazione dell'algoritmo. Questo potrebbe offrire miglioramenti significativi sia in termini di efficienza e permettere un miglioramento della precisione senza perdite di prestazioni.

5.3 Esperimento 2 - Variazione raggio dei candidati su GPU

In questo capitolo viene introdotto il secondo esperimento. L'obiettivo è quello di replicare l'analisi condotta nel primo, ma utilizzando l'algoritmo sviluppato specificamente per **GPU**. Questo ci permetterà di osservare eventuali miglioramenti a livello di tempo di esecuzione rispetto all'implementazione su **CPU**.

5.3.1 Analisi dell'ottimizzazione eseguita su GPU : Risultati e Conclusioni

La parallelizzazione ha portato a un notevole incremento delle prestazioni in termini di tempo di esecuzione. Nello specifico, abbiamo osservato:

- Un incremento della velocità di esecuzione di oltre il **300%** rispetto all'implementazione originale su CPU eseguendolo sul *laptop*.
- Un incremento della velocità di esecuzione di oltre il **400%** rispetto all'implementazione originale su CPU eseguendolo sulla *board*.
- Una stabilità nella **precisione dei dati**, l'accuratezza è rimasta pressochè invariata.

Questo significativo aumento di efficienza dimostra il potenziale della computazione parallela su GPU per questo tipo di algoritmi di ricerca intensiva. Di seguito sono presentati i risultati dell'esperimento, accompagnati da un'analisi comparativa delle prestazioni su **CPU**.

Risultati Laptop

Tabella 5.4: Risultati globali (raggio default 2) (**KITTY DATASET - 04**)

Device(laptop)	CPU	GPU	
Translation error	0.410	0.450	
Rotational error	0.220	0.220	
ATE (m)	0.610	0.480	
RPE (m)	0.014	0.015	
RPE (deg)	0.041	0.042	
ORB Extraction (ms)	5.108	5.123	
Stereo Matching (ms)	3.302	**0.939**	← miglioramento (351%)
ORB Extraction (σ)	3.425	3.388	
Stereo Matching (σ)	0.595	**0.117**	← miglioramento (508%)

Tabella 5.5: Risultati temporali (**KITTY DATASET - 04**) [*laptop*]

Radius multiplier	1.0	2.0	4.0	8.0	16.0
Stereo Matching CPU (ms)	3.022	3.391	3.962	4.835	6.345
Stereo Matching GPU (ms)	0.829	0.973	1.300	1.938	3.218
Stereo Matching CPU (σ)	0.543	0.630	0.688	0.748	0.685
Stereo Matching GPU (σ)	0.099	0.129	0.196	0.272	0.353

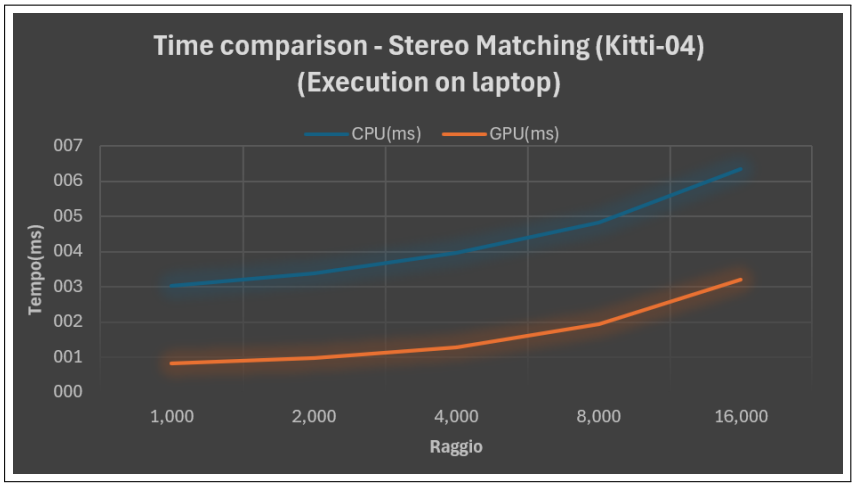

Figura 5.2: Comparazione dei tempi sull'algoritmo di Stereo Matching

Tabella 5.6: Risultati globali (raggio default 2) (**KITTY DATASET - 06**)

Device(laptop)	CPU	GPU	
Translation error	0.670	0.690	
Rotational error	0.300	0.360	
ATE (m)	4.510	4.850	
RPE (m)	0.014	0.015	
RPE (deg)	0.033	0.034	
ORB Extraction (ms)	4.729	4.768	
Stereo Matching (ms)	3.089	**0.947**	← miglioramento (326%)
ORB Extraction (σ)	3.580	3.399	
Stereo Matching (σ)	0.553	**0.124**	← miglioramento (445%)

Tabella 5.7: Risultati temporali (**KITTY DATASET - 06**) [*laptop*]

Radius multiplier	1.0	2.0	4.0	8.0	16.0
Stereo Matching CPU (ms)	2.833	3.231	3.732	4.679	6.282
Stereo Matching GPU (ms)	0.789	0.954	1.224	1.867	2.925
Stereo Matching CPU (σ)	0.535	0.557	0.646	0.717	0.682
Stereo Matching GPU (σ)	0.085	0.117	0.166	0.243	0.261

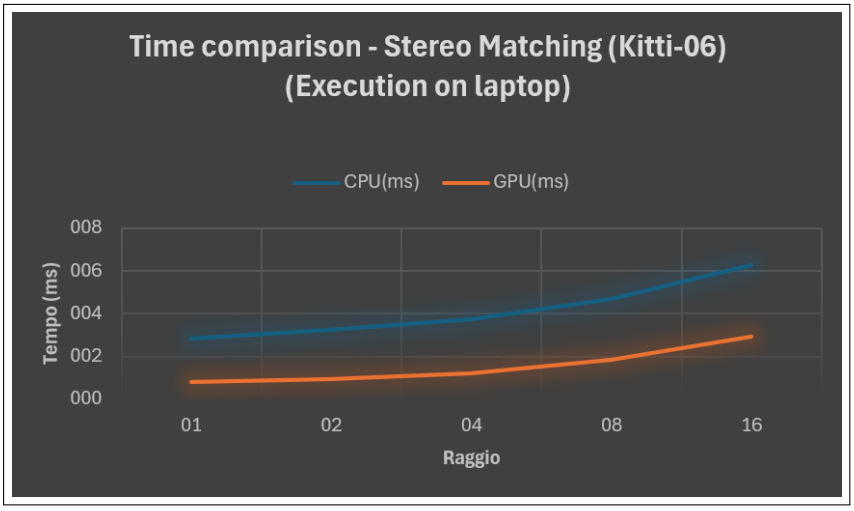

Figura 5.3: Comparazione dei tempi sull'algoritmo di Stereo Matching

Tabella 5.8: Risultati globali (raggio default 2) (**KITTY DATASET - 07**)

Device	CPU	GPU	
Translation error	0.540	0.480	
Rotational error	0.110	0.130	
ATE (m)	0.700	0.630	
RPE (m)	0.017	0.017	
RPE (deg)	0.034	0.032	
ORB Extraction (ms)	5.520	5.524	
Stereo Matching (ms)	3.284	**0.969**	← miglioramento (338%)
ORB Extraction (σ)	6.911	6.841	
Stereo Matching (σ)	0.545	**0.130**	← miglioramento (420%)

Tabella 5.9: Risultati temporali (**KITTY DATASET - 07**) [*laptop*]

Radius multiplier	1.0	2.0	4.0	8.0	16.0
Stereo Matching CPU (ms)	3.087	3.573	4.072	4.897	6.391
Stereo Matching GPU (ms)	0.791	0.972	1.253	1.905	3.069
Stereo Matching CPU (σ)	0.535	0.527	0.642	0.689	0.688
Stereo Matching GPU (σ)	0.091	0.127	0.161	0.307	0.404

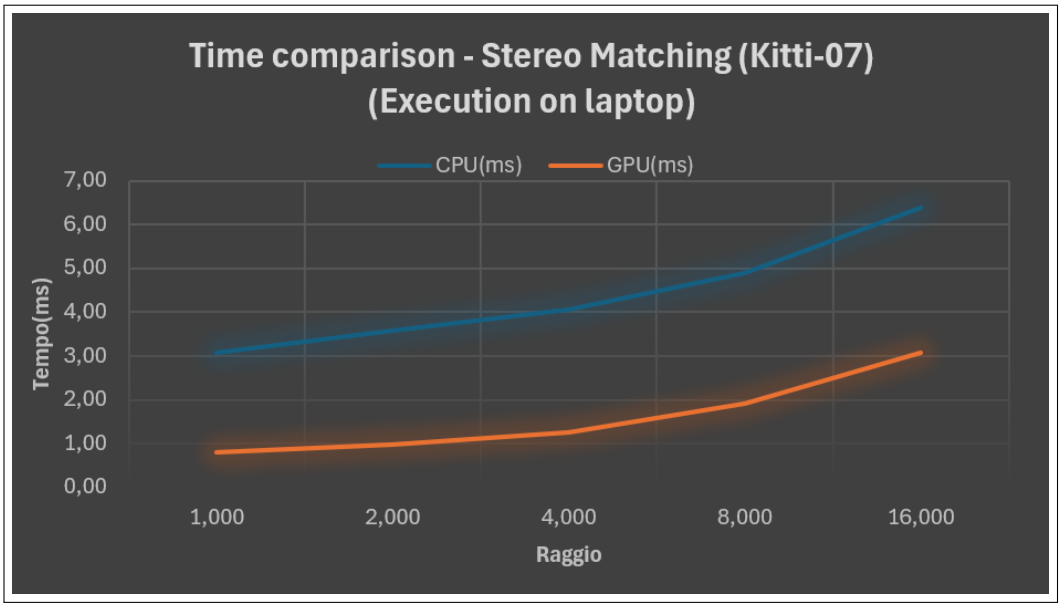

Figura 5.4: Comparazione dei tempi sull'algoritmo di Stereo Matching

Risultati Board

Tabella 5.10: Risultati temporali (**KITTY DATASET - 04**) [*board*]

Radius multiplier	1.0	2.0	4.0	8.0	16.0
Stereo Matching CPU (ms)	7.722	9.699	11.033	14.511	20.841
Stereo Matching GPU (ms)	2.046	2.309	2.381	3.238	4.347
Stereo Matching CPU (σ)	0.711	1.343	0.896	1.325	1.742
Stereo Matching GPU (σ)	0.241	0.340	0.506	0.629	0.681

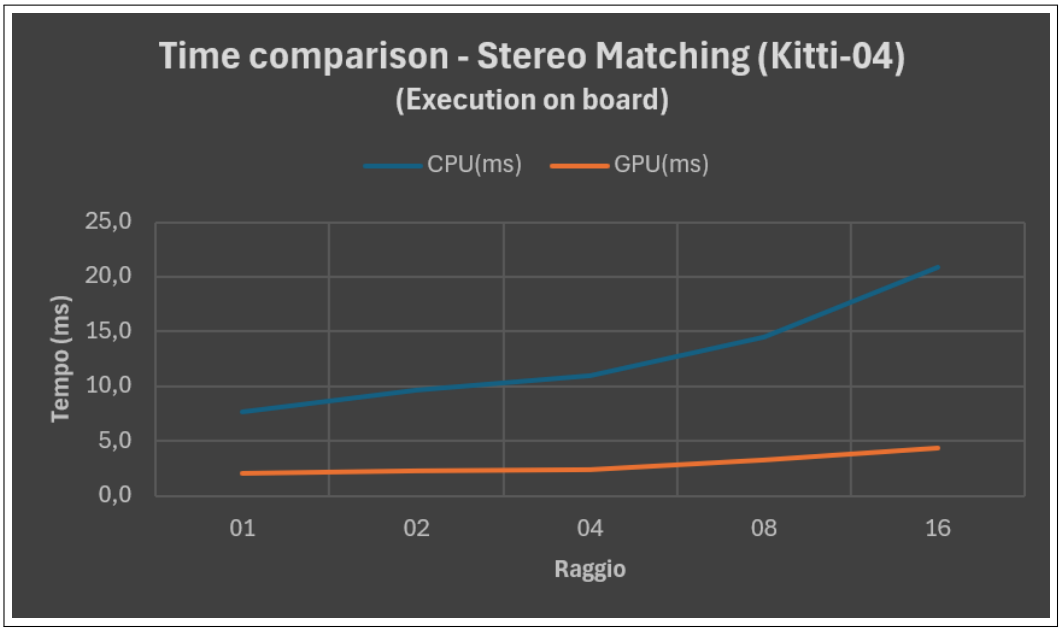

Figura 5.5: Comparazione dei tempi sull'algoritmo di Stereo Matching

Tabella 5.11: Risultati temporali (**KITTY DATASET - 06**) [*board*]

Radius multiplier	1.0	2.0	4.0	8.0	16.0
Stereo Matching CPU (ms)	7.275	8.454	10.920	13.454	19.630
Stereo Matching GPU (ms)	1.928	2.139	2.216	3.037	4.117
Stereo Matching CPU (σ)	0.771	1.005	1.106	1.023	1.432
Stereo Matching GPU (σ)	0.212	0.273	0.311	0.598	0.770

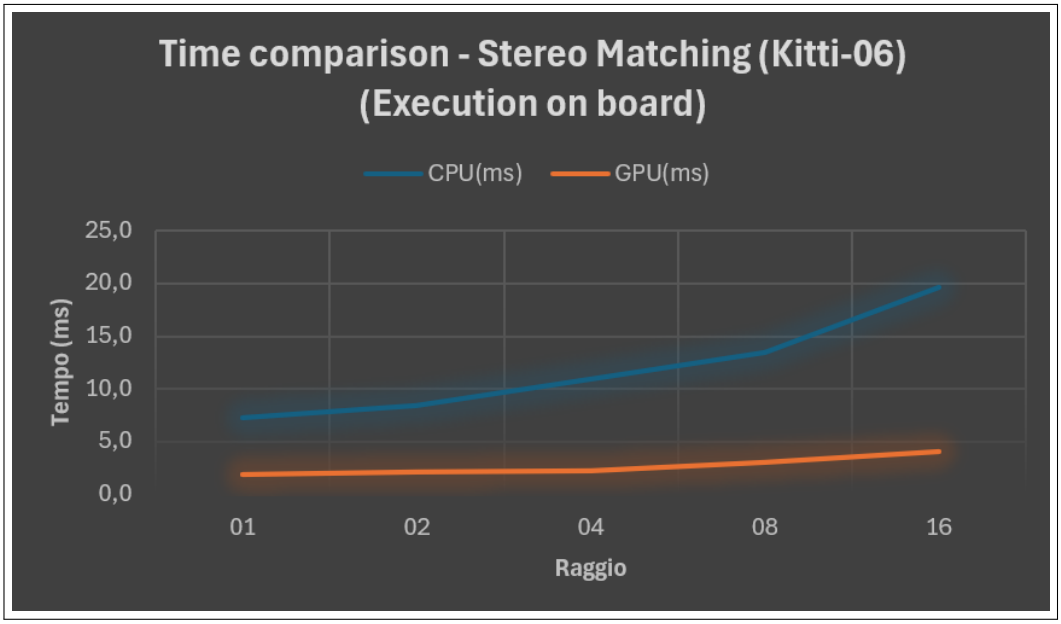

Figura 5.6: Comparazione dei tempi sull'algoritmo di Stereo Matching

Tabella 5.12: Risultati temporali (**KITTY DATASET - 07**) [*board*]

Radius multiplier	1.0	2.0	4.0	8.0	16.0
Stereo Matching CPU (ms)	7.706	8.906	11.149	14.446	20.450
Stereo Matching GPU (ms)	1.905	2.099	2.206	2.740	3.479
Stereo Matching CPU (σ)	0.758	0.749	0.954	1.096	1.504
Stereo Matching GPU (σ)	0.143	0.190	0.277	0.616	0.703

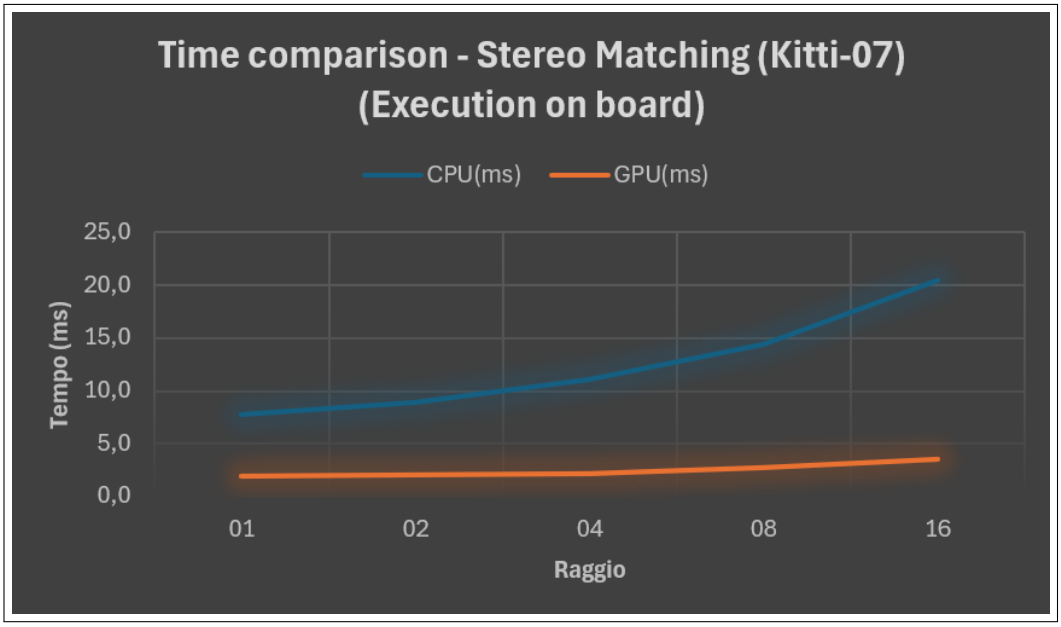

Figura 5.7: Comparazione dei tempi sull'algoritmo di Stereo Matching

Capitolo 6

Conclusioni

A seguito dell'implementazione delle ottimizzazioni descritte nello sviluppo di questo elaborato, si è riscontrato un notevole incremento delle prestazioni. Nello specifico, è stato registrato un miglioramento superiore al **300%**. Questo risultato rappresenta un passo significativo verso l'implementazione di sistemi **SLAM** più efficienti e adatti all'uso in tempo reale. Il miglioramento delle prestazioni ottenuto dimostra il potenziale del calcolo parallelo nel campo della visione artificiale e della robotica.

La flessibilità offerta dalla possibilità di regolare il raggio di ricerca apre nuove strade per l'adattamento del sistema a diverse condizioni operative, permettendo di trovare il giusto equilibrio tra velocità di elaborazione e precisione della localizzazione e mappatura.

Il notevole guadagno in termini di prestazioni offre l'opportunità di aumentare la complessità computazionale in futuro. Questo potrebbe permettere di ottenere risultati ancora più precisi **senza compromettere le prestazioni** complessive del sistema in termini di *precisione* e *tempo*. Ad esempio:

- Potremmo aumentare il numero di punti di interesse analizzati.
- Potremmo implementare algoritmi di matching più sofisticati e computazionalmente intensivi.
- Potremmo estendere l'analisi a più frame o a risoluzioni di immagine più elevate.

In conclusione, questo lavoro ha dimostrato il potenziale dell'utilizzo delle **GPU** per migliorare le prestazioni di **ORB-SLAM3**, aprendo la strada a

implementazioni più efficienti di sistemi SLAM in applicazioni che richiedono elaborazione in tempo reale.

Appendice A

Installazioni del software

A.1 Installazione del sistema operativo

Il sistema operativo scelto per il progetto è **Linux Mint 21.3**, è una distribuzione basata su *Ubuntu* che offre un ambiente desktop intuitivo e facile da usare. Tra i vantaggi principali di Linux Mint e di Linux in generale troviamo la stabilità, la flessibilità, e una comunità di supporto attiva. *Linux Mint*, in particolare, offre un'interfaccia utente simile a *Windows*, rendendo la transizione semplice per gli utenti che non hanno familiarità con altri sistemi Linux.

Figura A.1: Linux Mint - Logo

Di seguito sono elencati i passaggi principali per l'installazione del sistema operativo:

1. **Scaricare la distribuzione:** La distribuzione di Linux Mint 21.3 può essere scaricata direttamente dal sito ufficiale: https://linuxmint.com/download.php.
2. **Creare una chiavetta USB avviabile:** Per scrivere l'immagine del sistema operativo su una chiavetta USB, è stato utilizzato un software

come *Rufus*. La chiavetta è stata preparata con una partizione di tipo GPT e con il file system NTFS per garantire la compatibilità.

3. **Installazione in dual-boot:** L'installazione è stata effettuata in modalità *dual-boot*, permettendo così di mantenere il sistema operativo preesistente insieme a Linux Mint. Il dual-boot è una configurazione che consente di scegliere quale sistema operativo avviare all'accensione del computer, garantendo la massima flessibilità nell'uso del dispositivo.

4. **Modifica dell'ordine di boot:** Dopo aver creato la chiavetta USB, è stato necessario modificare l'ordine di avvio (boot) dal **BIOS/UEFI** del computer, per poi procedere con l'installazione definitiva di *Linux Mint*.

A.2 Installazione ed avvio di ORB-SLAM 3

In questo capitolo verrà descritta l'installazione e l'avvio del sistema ORB-SLAM3, uno dei più avanzati algoritmi di localizzazione e mappatura simultanea (SLAM). Verranno spiegati i passaggi principali necessari per configurare correttamente l'ambiente, installare le dipendenze e avviare ORB-SLAM3. L'obiettivo è fornire una guida dettagliata che permetta di eseguire l'algoritmo su un sistema basato su Linux, evidenziando eventuali problematiche comuni e come risolverle.

A.2.1 Installazione del CUDA Toolkit

Per poter sviluppare applicazioni con CUDA, è necessario installare il CUDA Toolkit, disponibile direttamente dal sito di NVIDIA. Questo toolkit contiene tutti gli strumenti necessari per la compilazione e l'esecuzione di codice parallelo su GPU, inclusi compilatori, librerie e campioni di codice. I passaggi principali necessari sono i seguenti e sono presi dal seguente tutorial https://docs.nvidia.com/cuda/cuda-installation-guide-linux/index.html

- 1. Controllare la versione di CUDA con il comando **nvidia-smi**
- 2. Scaricare la versione di CUDA corrispondente alla versione restituita dal comando precedente. Example:

  ```
  Driver Version: 535.171.04   CUDA Version: 12.2
  ```

- 3. Aprire il file .bashrc e inserire il seguente comando:

  ```
  export PATH=/usr/local/cuda-{version}/bin/:$PATH
  ```

A.2.2 Ottenimento del Kitty Data Set

Il **KITTI dataset** è un insieme di dati ampiamente utilizzato nella ricerca su visione artificiale e guida autonoma. È composto da immagini, informazioni di navigazione (GPS/IMU), e dati LiDAR raccolti da veicoli in ambienti urbani, utilizzato per allenare e valutare algoritmi come rilevamento di oggetti, segmentazione semantica, e SLAM (Simultaneous Localization and Mapping).

Per installarlo è sufficiente loggarsi sul sito https://www.cvlibs.net/datasets/kitti/eval_odometry.php e scaricare il pacchetto *odometry data set (gray-scale, 22 GB)*

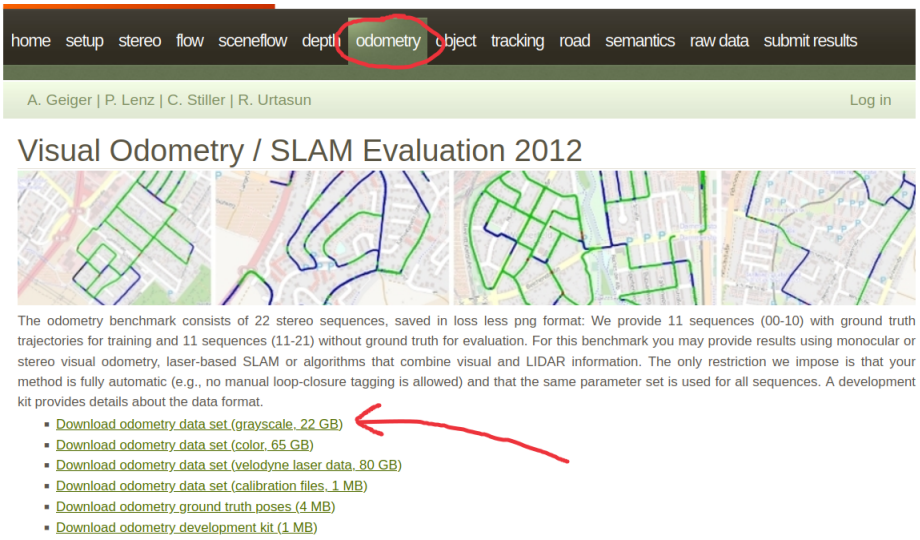

Figura A.2: Indicazione pacchetto da scaricare

A.2.3 Installazione definitiva di ORB-SLAM3

Per avvicinarci all'installazione finale di ORB-SLAM3 parallelo prima scarichiamo la versione non ottimizata, a seguire i passaggi descritti di seguito.

1. **Clonare il repository:** Per prima cosa, clonare il repository di ORB-SLAM3 dal seguente URL: https://github.com/UZ-SLAMLab/ORB_SLAM3. Utilizzare il seguente comando nel terminale:

```
git clone https://github.com/UZ-SLAMLab/ORB_SLAM3.git
```

2. **Eseguire la build:** Dopo aver clonato il repository, entrare nella directory di ORB-SLAM3 ed eseguire il processo di build con i comandi seguenti:

```
cd ORB_SLAM3
chmod +x build.sh
./build.sh
```

3. **Eseguire ORB-SLAM3:** Una volta completata la build, è possibile avviare ORB-SLAM3 con il comando seguente. Entrare nella directory principale di ORB-SLAM3 e utilizzare il comando:

```
./Examples/Stereo/stereo_kitti Vocabulary/ORBvoc.txt
Examples/Stereo/KITTIX.yaml ~/Desktop/dataset/sequences/00
```

   Questo comando avvierà ORB-SLAM3 utilizzando il KITTI dataset per la modalità stereo. In base alla sequenza che andremo ad utilizzare bisogna cambiare **KITTIX.yaml** :

   - KITTI00-02.yaml → sequenze 00,01,02.
   - KITTY03.yaml → sequenze 03.
   - KITTY04-12.yaml → sequenze 04,05,06,07,08,09,10,11,12.

Una volta che si è riusciti ad avviare la versione sopra citata, è possibile a passare alla versione parallela. Infatti la versione attualmente in esecuzione, non è quella citata all'inizio del capitolo, questo perchè non contiene la parte di *Estrazione ORB* parallelizata. Per fare in modo di eseguire la versione in parallelo è necessario seguire i medesimi step con il seguente repository: https://git.hipert.unimore.it/fmuzzini/cuda-accelerated-orb-slam

Bibliografia

[Babu, 2024] Babu, A. S. (2024). La slam da un punto di vista algoritmico (parte 2). https://www.digikey.it/it/blog/slam-from-an-algorithmic-point-of-view-part-2. Accessed: 11 Ottobre 2024.

[Ganti and Waslander, 2018] Ganti, P. and Waslander, S. (2018). Visual slam with network uncertainty informed feature selection.

[Gupta, 2020] Gupta, P. (2020). The cuda programming model. https://developer.nvidia.com/blog/cuda-refresher-cuda-programming-model/. Accessed: 27 September 2024.

[Kerbl et al., 2022] Kerbl, B., Michael, K., Martin, W., and Markus, S. (2022). Cuda tutorial: Part 1. https://cuda-tutorial.github.io/part1_22.pdf. Accesso il 12 giugno 2024.

[Miaoyu Cai, 2022] Miaoyu Cai (2022). A gpu-accelerated method for 3d nonlinear kelvin ship wake patterns simulation - scientific figure. https://www.researchgate.net/figure/Illustration-of-CUDA-execution-mode-and-thread-organization-hierarchy-23_fig5_375537168. Accessed: 25 September 2024.

[Mur-Artal and Tard´os, 2017] Mur-Artal, R. and Tard´os, J. D. (2017). Orb-slam2: an open-source slam system for monocular, stereo and rgb-d cameras. http://arxiv.org/pdf/1610.06475/. Accessed: 13 Giugno 2024.

[NVIDIA-website, 2023a] NVIDIA-website (2023a). Nvidia official website - jetson serie xavier (board). https://www.nvidia.com/it-it/autonomous-machines/embedded-systems/jetson-agx-xavier/. Accessed: 20 Ottobre 2024.

[NVIDIA-website, 2023b] NVIDIA-website (2023b). Nvidia official website - jetson serie xavier series. https://www.nvidia.com/it-it/

autonomous-machines/embedded-systems/jetson-xavier-series/. Accessed: 20 Ottobre 2024.

[Unknown(Blog), 2023] Unknown(Blog) (2023). Introduction to loop closure detection in slam. https://www.thinkautonomous.ai/blog/loop-closure/. Accessed: 11 Ottobre 2024.