Alma Mater Studiorum · Università di Bologna

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

PARALLELIZZAZIONE SU GPU ALGORITMO MARCHING SQUARES PER APPLICAZIONE INDUSTRIALE

Elaborato in:
High Performance Computing

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Correlatore:
Dott.
Matteo Roffilli

Presentata da:
Alessandro Sciarrillo

Anno Accademico 2022/2023

Indice

Capitolo 1

Abstract

Marching Squares(MS) é un algoritmo per la generazione di contorni in un campo scalare bidimensionale che viene ampiamente utilizzato nel Machine Vision in ambito industriale. Nella applicazione pratica in questione viene utilizzato su fotografie scattate da macchine per la selezione automatica della frutta per trovare i contorni di aree dell'immagine dove vengono riconosciuti dei difetti nel frutto. L'algoritmo viene applicato all'output di una CNN (Convolutional Neural Network) che é composto da una mappatura dei pixel dell'immagine in input nella rispettiva probabilità di appartenere ad una certa classe di difetto, vengono costruiti i contorni delle aree che hanno una probabilità maggiore di una certa soglia di contenere una certa classe. Le classi di difetto sono ad esempio: marcio, ruggine, danno da grandine fresca, danno da grandine cicatrizzato, danno da raccolta, danno da trasporto ecc..
Per ogni frutto che deve essere smistato correttamente dalle macchine in base alle sue condizioni vengono scattate più foto mentre viene trasportato su dei rulli che lo fanno roteare e permettono quindi alle fotocamere di raccogliere un insieme di scatti in cui il frutto é stato catturato in tutte le sue facce. Per ognuna delle foto scattate al frutto vengono generate delle matrici di probabilità per ogni classe di difetto, il risultato del processo di selezione é quindi l'insieme delle immagini dei vari lati di quel preciso frutto con i vari difetti racchiusi da un contorno che li identifica.
La costruzione di questo contorno viene attualmente effettuato da Python tramite il metodo `findContours` della libreria skimage che utilizza un'implementazione seriale dell'algoritmo Marching Squares, lo scopo di questa ricerca é di implementare una versione parallela su GPU dell'algoritmo in modo da ridurre i tempi di esecuzione che risultano un fattore di importanza fondamentale. Infatti ogni frazione di secondo risparmiata può essere utilizzata per aumentare il numero di frutti classificati in un'unità di tempo o per dedicare quel tempo ad altre elaborazioni utili a migliorare il risultato. Il metodo `findContours` di skimage é scritto in Python ma la parte principale in cui utilizza MS é stata scritta in Cython (codice Python-like che viene compilato in codice C) per migliorare i tempi di esecuzione, può essere quindi considerata come una versione seriale già

particolarmente ottimizzata.
L'obbiettivo è di parallelizzare proprio la stessa parte dell'algoritmo che skimage mantiene in Cython che è anche l'unica porzione di codice parallelizzabile dell'algoritmo MS.
Le principali strategie che verranno esplorate sono:

- utilizzo dell'ultima versione di nvc++ per la parallelizzazione in fase di compilazione del codice Cython
- utilizzo delle API Cuda-Python per il lancio di kernel Cuda (scritti manualmente) da Python

Il metodo migliore che verrà poi utilizzato per la soluzione finale sarà quello che sfrutta le API Cuda-Python e i kernel Cuda scritti manualmente, riuscirà infatti ad ottenere uno Speedup di circa x5**[NOTA: sulla mia macchina, attendo test su server azienda per aggiornare valore e inserire specifiche macchina] rispetto al corrispondente codice seriale della libreria skimage. Verrà anche analizzato l'overhead nel lancio dei kernel Cuda introdotto da Python rispetto a una versione scritta in C.
Nella soluzione finale viene inoltre implementata una elaborata versione parallela di exlusive scan composta da più kernel che risulta di particolare interesse nell'ambito dell'High Performance Computing.

Capitolo 2

Introduzione alla programmazione Parallela

Capitolo 3

Introduzione e Analisi del Problema

3.1 Introduzione

3.1.1 Marching Squares (MS)

L'algoritmo Marching Squares genera contorni per un campo scalare a due dimensioni, data una matrice di valori e una soglia é in grado di trovare un insieme di segmenti che delimitano le aree della matrice in cui il valore contenuto dalle singole celle é maggiore della soglia data.
Una delle elaborazioni più utilizzate viene effettuata considerando separatamente ogni gruppo di quattro elementi della matrice disposti a forma di quadrato, ognuno di questi quadrati può ricadere in uno di sedici diversi casi possibili ben definiti. Per definire a quale tipo appartiene un certo quadrato bisogna prima binarizzare i valori dei quattro spigoli in base alla soglia data, la posizione dei valori negli spigoli é importante poiché i sedici casi sono definiti con un'orientazione ben precisa. Nella figura 3.1 é possibile vedere una generica rappresentazione dei sedici casi possibili nella versione più comune di Marching Squares. In questa versione vengono considerate delle Isolinee ma esiste anche una variante in cui vengono considerate delle Isobande che sono costruite con l'aggiunta alle barre di contorno di upper e lower thresholds come rappresentato nell'immagine 3.2. Esistono anche versioni che invece dei quadrati utilizzano triangoli e vengono applicate per l'individuazione di meshes triangolari.

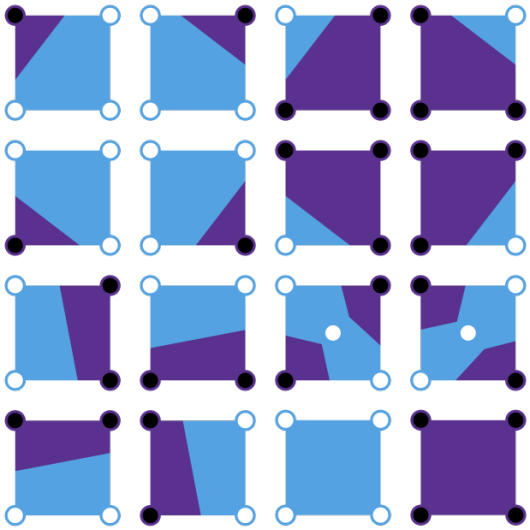

Figura 3.1: Sedici casi possibili in cui possono ricadere i quadrati composti dai quattro valori.

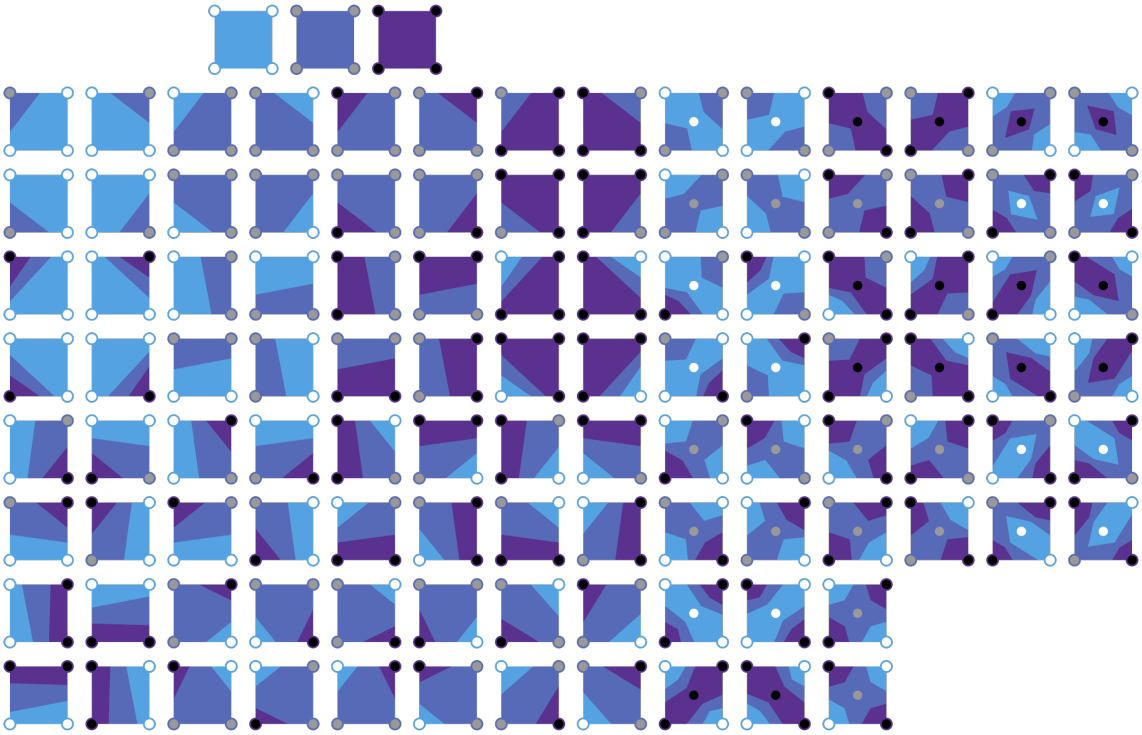

Figura 3.2: Casi possibili nella versione con isobande in cui possono ricadere i quadrati composti dai quattro valori.

L'algoritmo é embarrassingly parallel per quanto riguarda la classificazione per tipo di ogni cella (quadrato con valori negli spigoli) poiché puó essere svolta in modo indipendente tra le celle.
La fase di ricostruzione dei contorni invece può essere svolta sia in parallelo che in seriale utilizzando tecniche e modalità differenti che dipendono dall'utilizzo finale a cui il codice é destinato.
MS è utilizzato per molte applicazioni pratiche in settori di particolare interesse come ad esempio:

- Computer Graphics per generare immagini 3D da dati 2D.
- Rilevamento remoto in immagini satellitari o radar.
- Medicina per analizzare scansioni CT o immagini MRI dove possono essere identificare anomalie come tumori.
- Scienze naturali per l'analisi di dati meteorologici e oceanici nell'identificazione di aree di pioggia o di correnti forti.
- Cartografia per la generazione di mappe relative a paesi o città da dati 2D.

3.1.2 Problema Reale

Bioretics è l'azienda con cui è stara svolta la ricerca, uno dei settori in cui opera è quello della selezione automatica della frutta. La selezione della frutta è un processo svolto in questo caso da macchine dotate di rulli e fotocamere, i frutti entrano nella macchina all'interno di tazze e vengono fatti roteare da dei rulli in modo da poter acquisire con delle camere fissate all'interno della macchina delle immagini di tutta la superficie dei frutti. Le immagini scattate per ogni frutto vengono processate e passate ad una CNN che restituisce delle matrici della stessa dimensioni delle immagini scattate, una per ogni classe di difetto che si vuole valutare, che hanno come valore la probabilità che il rispettivo pixel appartenga a quella classe.

L'azienda offre in sostanza un prodotto software che viene eseguito da macchine per la selezione della frutta e include l'utilizzo dell'algoritmo Marching Squares (MS). La sfida proposta dall'azienda è quella di ridurre i tempi di esecuzione di MS che è utilizzato nella fase di segmentazione dei difetti. Le macchine per la selezione gestiscono un flusso di circa 10 frutti al secondo, ne consegue che ci sia approssimativamente 0.1s a disposizione per ogni frutto. Quindi un'implementazione parallela dell'algoritmo MS, che riesca a ottenere uno speedup anche solo di 1.1 sarebbe considerato un risultato positivo per l'azienda. Scendendo più nel merito degli aspetti tecnici, all'interno delle macchine vengono scattate immagini dei frutti da camere fissate e calibrate che sono poi

elaborate da una CNN che a sua volta restituisce un tensore WxHxC, dove ogni canale rappresenta una classe (esempio: picciolo, ammaccatura, muffa). I canali vengono poi passati singolarmente al MS che definisce i contorni di ogni classe in base ad un valore di soglia specificato. Il risultato del processo è visibile dall'immagine della figura 3.3 dove si possono notare le varie classi delimitate da colori differenti.

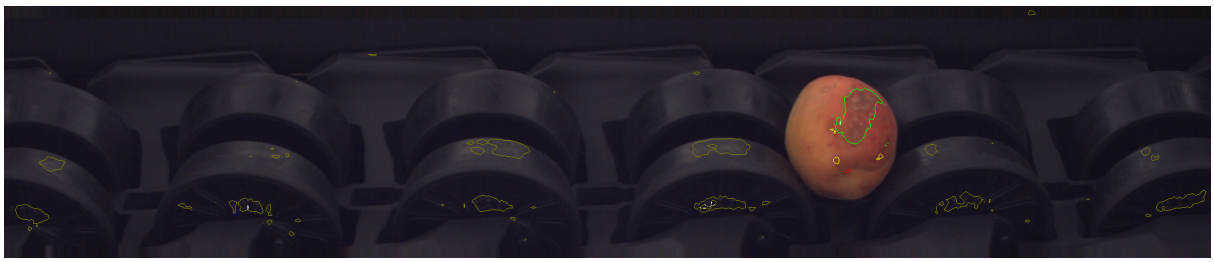

Figura 3.3: Immagine scattata da macchina per la selezione della frutta senza elaborazioni applicate.

L'implementazione di MS che è attualmente utilizzata dall'azienda deriva dalla libreria scikit-image che offre una versione seriale dell'algoritmo. Il metodo della libreria che viene chiamato è scritto in Python ma la parte principale è stata scritta in Cython. Il codice in Cython deve essere compilato prima di essere eseguito, nel complesso però riduce i tempi di esecuzione rispetto all'equivalente in Python. Bisogna quindi considerare che il tempo totale di esecuzione di MS è stato già in parte ridotto dagli autori della libreria.
In tutte le macchine sulle quali esegue il codice dell'azienda sono montate schede video di fascia alta per quanto riguarda le prestazioni, l'utilizzo di codice parallelo su GPU può essere quindi ampiamente sfruttato per ridurre i tempi di esecuzione e sollevare del carico la CPU su cui altrimenti ricadrebbe con esecuzioni seriali che in confronto sono estremamente dispendiose in termini di tempo.

3.1.3 Obbiettivo

L'obbiettivo concordato con l'azienda è quello di implementare una versione parallela su GPU (scheda video) dell'algoritmo Marching Squares, l'ottenimento di uno speedup rispetto alla versione utilizzata attualmente ovvero il metodo `findContours` della libreria skimage sarebbe considerata un successo.
Il software dell'azienda che attualmente richiama la funzione `findContours` è scritta in Python, è necessario quindi riuscire trovare un metodo per poter sfruttare l'esecuzione parallela su GPU da Python, operazione non comune dato che solitamente i kernel Cuda sono lanciati da codice C o C++ ovvero a un livello di astrazione molto più basso di Python e con strutture dati come puntatori compatibili con quelli di Cuda.

L'implementazione finale può essere rappresentata dallo schema in figura 3.4 ovvero una componente software che può essere richiamata direttamente da codice Python cioè un altro componente Python e delle parti aggiuntive di codice più a basso livello che siano in grado di lanciare ed eseguire codice sulla GPU in parallelo. L'unica parte della soluzione per cui la scelta del linguaggio da utilizzare risulta automatica è quella con cui si interfaccerà il codice dell'azienda, sarà quindi un modulo Python da cui poi si cercherà una strategia per arrivare all'esecuzione su GPU.

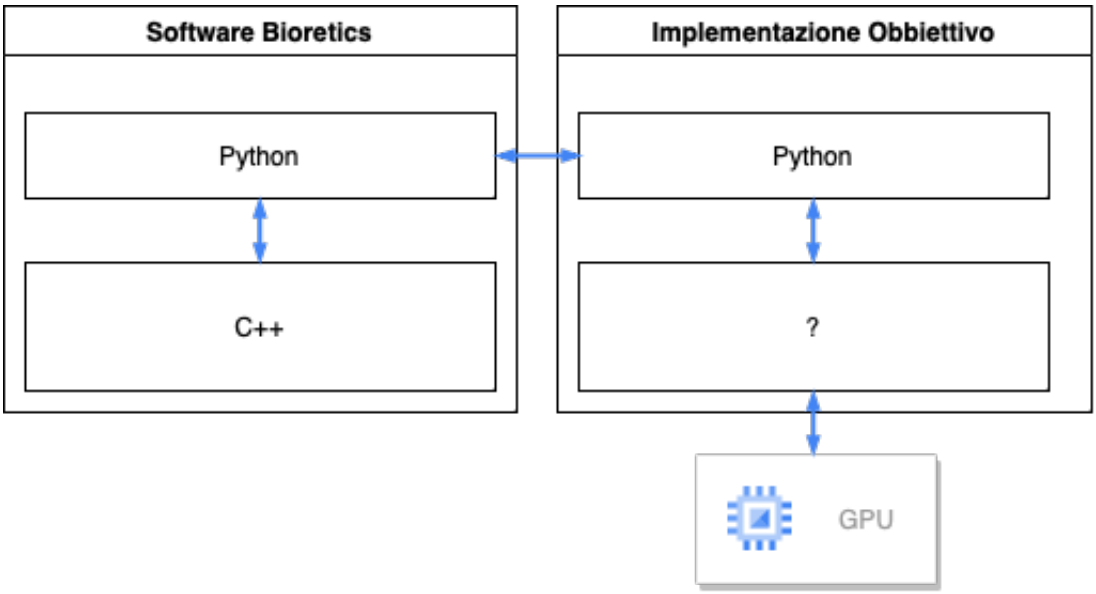

Figura 3.4: Schema linguaggi utilizzati dal software dell'azienda e da utilizzate per la soluzione finale.

3.2 Analisi implementazione skimage

Il metodo `findContours` di skimage composto da due fasi principali, nella prima viene richiamato un metodo `_get_contour_segments` che trova le coordinate dei segmenti che costituiscono i contorni e una seconda parte in cui viene richiamato `_assemble_contours` che dall'output di `_get_contour_segments` unisce i segmenti che si intersecano sui bordi delle celle formando linee spezzate.
Il metodo `_get_contour_segments` è contenuto nel file `_find_contours_cy.pyx` che è scritto in codice Cython, questo linguaggio è un superset di Python che permette di scrivere codice Python-like che effettua chiamate a funzioni C e può dichiarare tipi di dato del C, queste caratteristiche permettono al compilatore di generare codice ottimizzato per quanto riguarda i tempi di esecuzione rispetto all'equivalente in Python.

Non è semplice stabilire lo speedup del codice Cython rispetto al Python poiché il confronto dipende fortemente dalle strutture dati usate, il numero di funzioni C e python richiamate e altri fattori che influiscono in maniera significativa sulla differenza tra i tempi di esecuzione dei due linguaggi. Indubbiamente le chiamate a funzioni C da codice Cython introducono un overhead rispetto alla stessa chiamata effettuata da C e le funzioni Python impiegano un tempo sicuramente al massimo uguale e sicuramente non minore delle stesse funzioni richiamate da codice Python nativo.
Nel complesso però se il codice Cython viene scritto con particolari accortezze nell'utilizzo di soli tipi di dato C e richiamando funzioni Python solo se indispensabili il codice che ne deriva risulta estremamente più veloce della versione Python, con tempi di esecuzione si avvicinano alla versione C che teoricamente può essere considerata un suo lower bound. Queste osservazioni sono riscontrabili graficamente nei risultati dei test generici riportati nella figura 3.5.

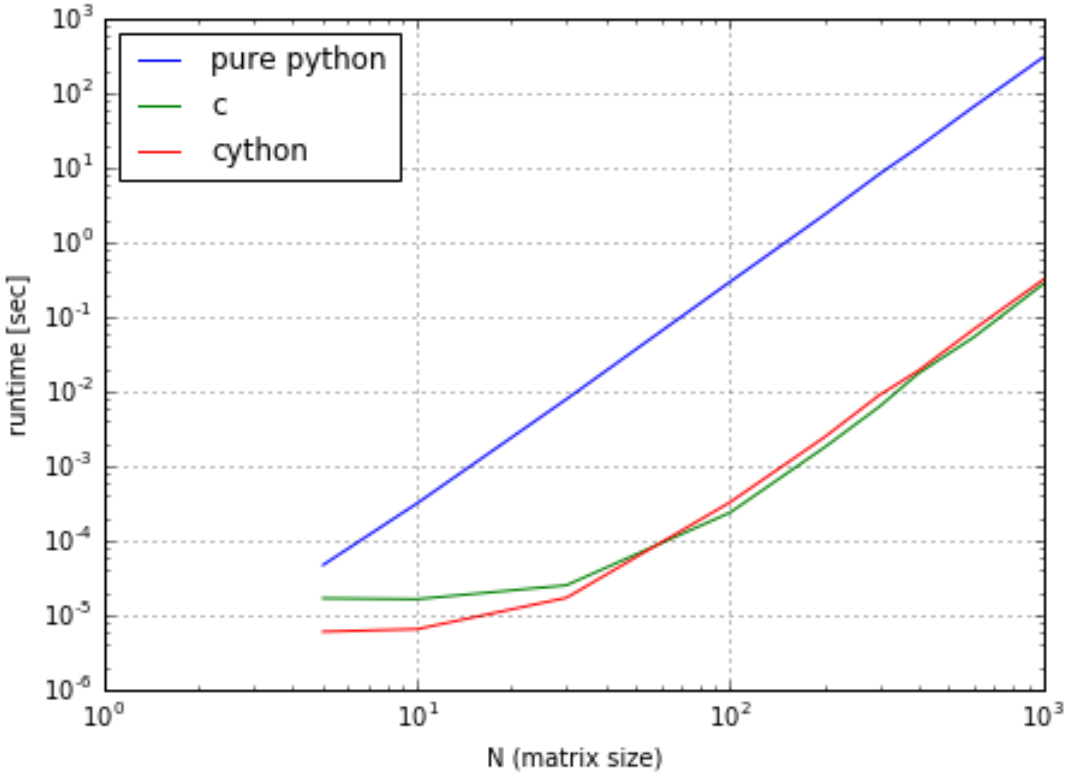

Figura 3.5: Grafico che mette in comparazione i tempi di esecuzione di Pyhon, C e Cython su un task di esempio.

Il codice del metodo Cython `_get_contour_segments` risulta particolarmente ottimizzato in quanto utilizza unicamente variabili di tipo C e richiama una sola funzione di Python ovvero `append` che può essere eseguita solitamente 0, 1 o al massimo 2 volte per ogni quadrato considerato. Il codice del metodo findContours può essere considerato quindi già particolarmente ottimizzato per essere un metodo Python, in quanto il suo tempo di esecuzione si può avvicinare molto ad una versione scritta in C.
Con una immagine 511x95 il metodo findContours impiega in media circa `0.0024s`**[NOTA: sulla mia macchina, attendo test su server azienda per aggiornare valore e inserire specifiche macchina] ovvero neanche tre millesimi di secondo.

3.3 Limiti della programmazione parallela

Nelle architettura Von Neumann la latenza introdotta dagli accessi alla memoria sono ordini di grandezza maggiori rispetto a un ciclo di clock e creano bottleneck che influiscono molto sui tempi di esecuzione di una versione di codice parallela. Nell'applicazione pratica proposta dall'azienda le immagini da elaborare sono di dimensioni 511x95, con una versione parallela del Marching Squares idealmente ogni Cuda Core si occuperebbe di un singolo quadrato, con immagini di queste dimensioni servirebbero 47940 esecuzioni (ovvero il numero di quadrati) che una GPU di fascia alta riesce a coprire utilizzando tutti i suoi Cuda Core in parallelo reiterando il procedimento qualche volta.
Indipendentemente da come può essere gestito il lancio e l'esecuzione del MS su GPU è indispensabile che l'immagine sia letta dalla memoria principale della macchina, caricata sulla memoria della GPU e in seguito il processo opposto sul risultato ottenuto sulla GPU. Queste operazioni di trasferimento dati tra memorie sono estremamente dispendiose paragonate ai tempi di esecuzione delle istruzioni che un kernel Cuda può impiegare, questo rischia di essere uno dei maggiori problemi e limiti da affrontare per raggiungere uno speedup in quanto sicuramente una versione di codice parallelo ha il potenziale di essere più veloce del rispettivo seriale, ma al tempo del codice parallelo va sommato il tempo per caricare e scaricare i dati dalla GPU che influiranno in gran parte sul tempo di esecuzione finale.
Se l'immagine fosse di dimensioni maggiori il tempo dovuto al bottleneck potrebbe essere ammortizzato maggiormente e la versione parallela avrebbe più margine su quella seriale, nel nostro caso invece la grandezza dell'immagine non richiede un tempo di esecuzione sufficientemente grande da far passare in secondo piano quello dei trasferimenti tra memorie.
Il caso peggiore che si può presentare è che il rapporto dati da trasportare e le operazioni da effettuarci sia così sbilanciato che solamente il tempo di upload e download dei dati dalla GPU senza neanche contare il tempo di esecuzione del codice parallelo sia maggiore del tempo di esecuzione della versione seriale.

3.4 Analisi versione seriale di MS

La libreria `scikit-image` (skimage) è un pilastro per quanto riguarda l'elaborazione di immagini in ambito Open-Source, è largamente utilizzata sia in piccoli progetti che in contesti industriali. Il codice delle funzioni più utilizzate è particolarmente solido e ottimizzato pur mantenendo un'ampia compatibilità con una vasta lista di ambienti, tra queste funzioni ricade anche findContours che infatti presenta ottime prestazioni per essere una funzione seriale lanciata da Python.
Dal codice di findContours che è riportato nella sezione di codice 1, proveniente dal file `scikit-image/skimage/measure/_find_contours.py`, si può notare facilmente che vengono richiamati due metodi `_get_contour_segments` e `_assemble_contours` che costituiscono le due fasi di costruzione del risultato, la prima di pura ricerca algoritmica dei segmenti che costituiscono i contorni e la successiva di congiunzione dei contorni confinanti in linee spezzate.

Codice 1: Metodo find_contours di scikit-image

```
1  def find_contours(image, level=None, fully_connected='low',
2                    positive_orientation='low', *, mask=None):
3
4      ...
5      # parameters configuration
6      ...
7
8      segments = _get_contour_segments(
9          image.astype(np.float64),
10         float(level),
11         fully_connected == 'high',
12         mask=mask)
13     contours = _assemble_contours(segments)
14     if positive_orientation == 'high':
15         contours = [c[::-1] for c in contours]
16     return contours
```

Il metodo `_get_contour_segments` non è compreso nello stesso modulo di `find_contours` ma nel file `scikit-image/skimage/measure/_find_contours_cy.pyx` che come si può notare termina con `.pyx`, estensione che contraddistingue file contenenti codice Cython. Come già accennato infatti `_get_contour_segments` è stato scritto in Cython poiché è la parte computazionalmente più impegnativa dell'intero metodo `find_contours`.
La maggior parte dei tipi di dati sono derivati dal C++ e altri dal Python come riportato nel seguente estratto di codice 2 del metodo Cython.

Codice 2: Estratto dal Metodo _get_contour_segments di scikit-image

```
1  cimport numpy as cnp
2  cnp.import_array()
3  cdef extern from "numpy/npy_math.h":
4      bint npy_isnan(cnp.float64_t x)
5  cdef list segments = []
6  cdef bint use_mask = mask is not None
7  cdef unsigned char square_case = 0
8  cdef tuple top, bottom, left, right
9  cdef cnp.float64_t ul, ur, ll, lr
10 cdef Py_ssize_t r0, r1, c0, c1
```

L'utilizzo dei questi tipi di dato C++ rispetto ai tipi Python permette di ottenere una compilazione più aderente alle necessità e volontà del programmatore che può specificare in modo preciso i tipi di dato di cui necessita.
La porzione di codice più importante del metodo `_get_contour_segments` è costituita da due cicli for annidati che scorrono tutta la matrice in input e considerano un quadrato di 4 celle ad ogni iterazione, calcolano a quale dei sedici diversi tipi appartiene e aggiornano una struttura dati contenente tutti segmenti dei contorni trovati.
Nell'aggiornamento dei segmenti trovati possono presentarsi tre diversi casi per quanto riguarda il numero di segmenti trovati:

- 0 segmenti da aggiungere: per i tipi 0 e 15 che sono quadrati con i 4 spigoli rispettivamente sotto e sopra la soglia di threshold non è necessario aggiungere alcun segmento poiché la relativa sezione dell'immagine in questione è completamente esclusa o inclusa in una certa classe, non è attraversata quindi da segmenti del contorno come si può vedere dalla figura 3.6.

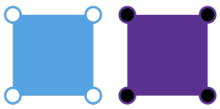

Figura 3.6: Caso 0 e 15 dell'algoritmo Marching Squares.

- 2 segmenti da aggiungere: per i tipi 6 e 9 che sono quadrati con le due coppie di spigoli opposti in cui una è maggiore e una minore della soglia. In questi due casi i segmenti da disegnare sono due, ci sono due aree appartenenti ad una certa classe che hanno contorni vicini ma separati come osservabile nella figura 3.7.

Figura 3.7: Caso 6 e 9 dell'algoritmo Marching Squares.

- 1 segmento da aggiungere: tutti gli altri tipi esclusi 0, 15, 6 e 9 hanno solamente un segmento che li attraversa in diverse modalità con diverse configurazioni di valori per i quattro spigoli del quadrato. Rappresentano i casi più comuni di aree di contorno e solitamente compongono la porzione maggiore di segmenti di cui sono costituite le linee spezzate con cui viene disegnato il contorno. Le 12 configurazioni sono rappresentate nella figura 3.8.

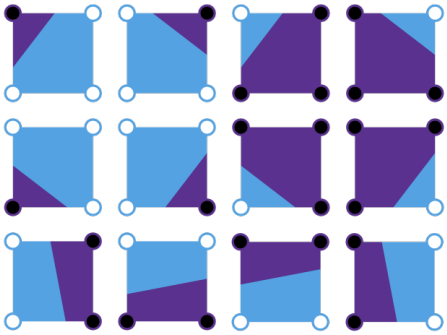

Figura 3.8: Casi 1-5,7,8,10-14 dell'algoritmo Marching Squares.

Ad ogni iterazione una volta definito a quale tipo appartiene un quadrato vengono aggiunti alla lista `segments` i relativi segmenti trovati che possono essere 0, 1 o 2 come appena specificato. Per ogni segmento trovato viene aggiunta a `segments` una tupla contenente altre due tuple in cui sono memorizzate le coordinate x e y del punto di inizio e fine del segmento, nel seguente estratto del metodo `_get_contour_segments` (Codice 3) viene riportata l'assegnazione delle coordinate per i punti che potranno essere aggiunti alla lista come estremi dei segmenti individuati.

Codice 3: Estratto dal Metodo _get_contour_segments di scikit-image

```
cdef inline cnp.float64_t _get_fraction(cnp.float64_t from_value,
                                        cnp.float64_t to_value,
                                        cnp.float64_t level):
    if (to_value == from_value):
        return 0
    return ((level - from_value) / (to_value - from_value))

top = r0, c0 + _get_fraction(ul, ur, level)
bottom = r1, c0 + _get_fraction(ll, lr, level)
left = r0 + _get_fraction(ul, ll, level), c0
right = r0 + _get_fraction(ur, lr, level), c1
```

Ognuno dei 16 casi deve essere gestito separatamente poiché le loro composizioni sono eterogenee per quanto riguarda i valori sugli spigoli e le coordinate da salvare. I segmenti vengono aggiunti alla lista `segments` tramite la funzione `append` che permette di costruire la lista di segmenti finale in modo incrementale senza dover conoscere a priori quale sarà la lunghezza finale della lista o quanti segmenti sarà necessario scrivere per ogni quadrato. Per i casi 0 e 15 non è necessario aggiungere nessun segmento a segments quindi si passa direttamente all'iterazione successiva, per gli altri casi invece si aggiungono 1 o 2 segmenti in base al caso in cui si ricade.

Estratto dal Metodo _get_contour_segments di scikit-image

```
# Manage case 0 and 15
if square_case in [0, 15]:
                continue

# Example for case 1-5, 7, 8, 10-14
if (square_case == 1):
                # top to left
                segments.append((top, left))
# Example for case 6 and 9
elif (square_case == 6):
                 segments.append((left, top))
                 segments.append((right, bottom))
```

3.4.1 Predisposizione alla parallelizzazione

L'algoritmo Marching Squares essendo embarassingly parallel è teoricamente predisposto ad una parallelizzazione, le soluzioni parallele però hanno necessità specifiche che il codice seriale non ha bisogno di rispettare, per riuscire a raggiungere questi requisiti partendo dal codice di `_get_contour_segments` è necessario effettuare radicali modifiche alle strutture dati utilizzate e al loro utilizzo.
La principale criticità riscontrabile nel metodo è l'utilizzo di una lista e del metodo `append` che non è possibile trasformare direttamente in ad esempio codice di un kernel Cuda poichè la dimensione finale della lista non è nota a priori e il metodo `append` non è disponibile in Cuda. La funzione `append` non potrebbe neanche essere utilizzata in una versione parallela ottimizzata poiché implica un accesso serializzato alla struttura dati lista per aggiungere il nuovo elemento in una nuova e ultima posizione della lista, questa operazione non può quindi essere effettuata contemporaneamente da più Cuda Core. Per una versione parallela su GPU dato che il tipo dei quadrati e l'aggiunta dei segmenti verrebbe eseguita contemporaneamente sarebbe necessario avere un accesso indipendente e non vincolato alla risorsa di output su cui scrivere, questo comporta la necessità di una struttura dati con un numero di posizioni già definito ma anche il numero di segmenti che ogni Cuda Core (ovvero per ogni quadrato valutato) necessita di scrivere.

Capitolo 4

Versione parallela con nvc++

La prima strategia esplorata consiste nel compilare il codice Cython con le ultime versioni del compilatore nvc++ che supportano dei flag per la parallelizzazione automatica su GPU. Nvidia ha recentemente aggiornato `stdpar`, un flag che promette di riuscir a parallelizzare automaticamente codice C++ su GPU utilizzando il compilatore nvc++.
Il processo parte da del codice Cython che tramite il proprio compilatore viene trasformato in codice C++, a questo punto si compila il codice C++ utilizzando nvc++ e una serie di flag per la parallelizzazione parallela compreso `stdpar`.
Il processo completo documentato da Nvidia è descritto nella figura 4.1.

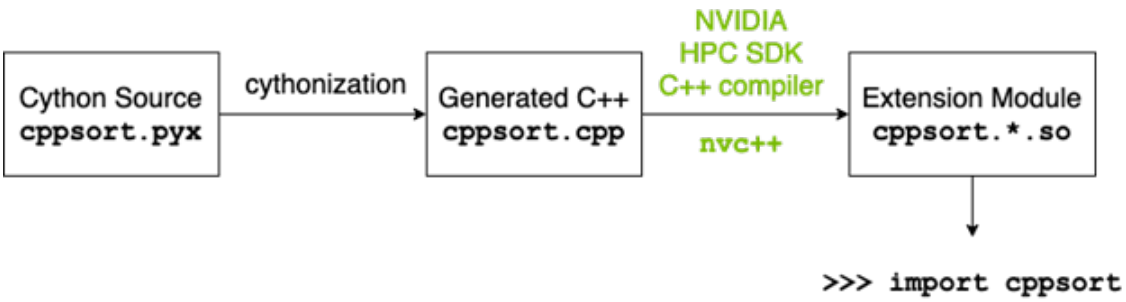

Figura 4.1: Schema di descrizione processo per utilizzo nvc++ in parallelizzazione codice Cython prodotto a Nvidia.

4.1 Risultati ottenuti

Capitolo 5

Versione parallela con API Cuda-Python

5.1 Risultati ottenuti

Capitolo 6

Risultati a Confronto

Capitolo 7

Conclusioni