

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

**Presentata da:**  
Alessandro Sciarrillo

**Correlatore:**  
Dott.  
Matteo Roffilli

**Anno Accademico 2022/2023**

# Indice

<b>1</b>	<b>Introduzione e Analisi del Problema</b>	<b>1</b>
1.1	Introduzione . . . . .	1
1.1.1	Marching Squares (MS) . . . . .	1
1.1.2	Problema Reale . . . . .	6
1.1.3	Obiettivo . . . . .	7
1.2	Analisi implementazione skimage . . . . .	8
1.3	Introduzione architettura GPU . . . . .	10
1.4	Limiti della programmazione parallela . . . . .	12
1.5	Analisi versione seriale di MS . . . . .	13
1.5.1	Predisposizione alla parallelizzazione . . . . .	17
<b>2</b>	<b>Versione parallela con nvc++</b>	<b>18</b>
2.1	Requisiti utilizzo nvc++ . . . . .	19
2.2	Potenziali risultati . . . . .	20
2.3	Marching Squares con nvc++ e -stdpar . . . . .	22
2.4	Problemi riscontrati . . . . .	23
<b>3</b>	<b>Versione parallela con API CUDA-Python</b>	<b>24</b>
3.1	API Cuda-Python . . . . .	24
3.1.1	CUDA Python workflow . . . . .	24
3.1.2	Utilizzo funzioni principali . . . . .	26
3.1.3	Prestazioni dichiarate . . . . .	31
3.2	Progettazione struttura codice parallelo . . . . .	32
3.2.1	Strutture dati risultato . . . . .	33
3.3	Kernel Cuda . . . . .	35
3.3.1	Kernel required_memory . . . . .	35
3.3.2	Kernel reduce . . . . .	37
3.3.3	Kernel exclusive scan (prescan) . . . . .	39
3.3.4	Kernel marching_squares . . . . .	42
3.4	Codice Python con API CUDA Python . . . . .	46

<b>4</b>	<b>Risultati ottenuti</b>	<b>49</b>
4.1	Immagine reale . . . . .	49
4.2	Immagine artificiale . . . . .	55
<b>5</b>	<b>Conclusioni</b>	<b>57</b>

# Abstract

Marching Squares (MS) è 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 è 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, permettendo alle fotocamere di raccogliere un insieme di scatti in cui il frutto è 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 è 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 `find_contours` della libreria skimage che utilizza un'implementazione seriale dell'algoritmo Marching Squares. Lo scopo di questa ricerca è 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 `find_contours` di skimage è scritto in Python ma la parte principale in cui utilizza MS è 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'obiettivo 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 2.6 volte rispetto al corrispondente codice seriale della libreria skimage.

Per la soluzione finale verranno progettati vari kernel CUDA da utilizzare in sequenza ottenendo un nuovo pattern di programmazione parallela che risulta di particolare interesse nell'ambito dell'High Performance Computing.

# Capitolo 1

## Introduzione e Analisi del Problema

### 1.1 Introduzione

#### 1.1.1 Marching Squares (MS)

L'algoritmo Marching Squares genera contorni per un campo scalare a due dimensioni. Data una matrice di valori e una soglia è in grado di trovare un insieme di segmenti che delimitano le aree della matrice in cui il valore contenuto dalle singole celle è 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 come illustrato nella figura 1.1; la posizione dei valori negli spigoli è importante poiché i sedici casi sono definiti con un'orientazione ben precisa.

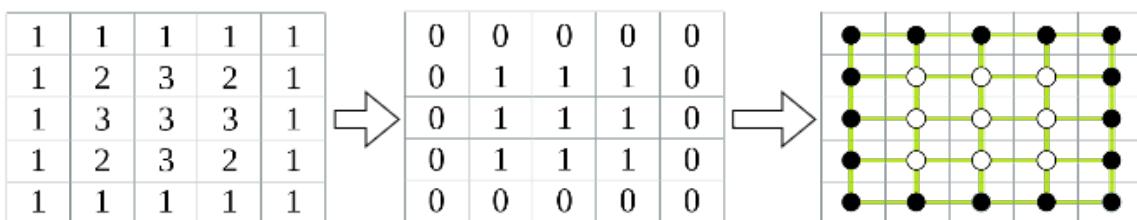


Figura 1.1: Schema processo di binarizzazione.

In seguito alla fase iniziale di binarizzazione, in base alla soglia dei valori di ogni pixel viene applicata la logica dell'algoritmo Marching Squares. Come primo passaggio si assegna a ogni cella un numero da 0 a 15 in base ai valori binari assunti dai suoi spigoli, in seguito ogni cella viene mappata con il numero ad essa assegnato e una look-up table nella rispettiva disposizione di segmenti come mostrato nella figura 1.2.

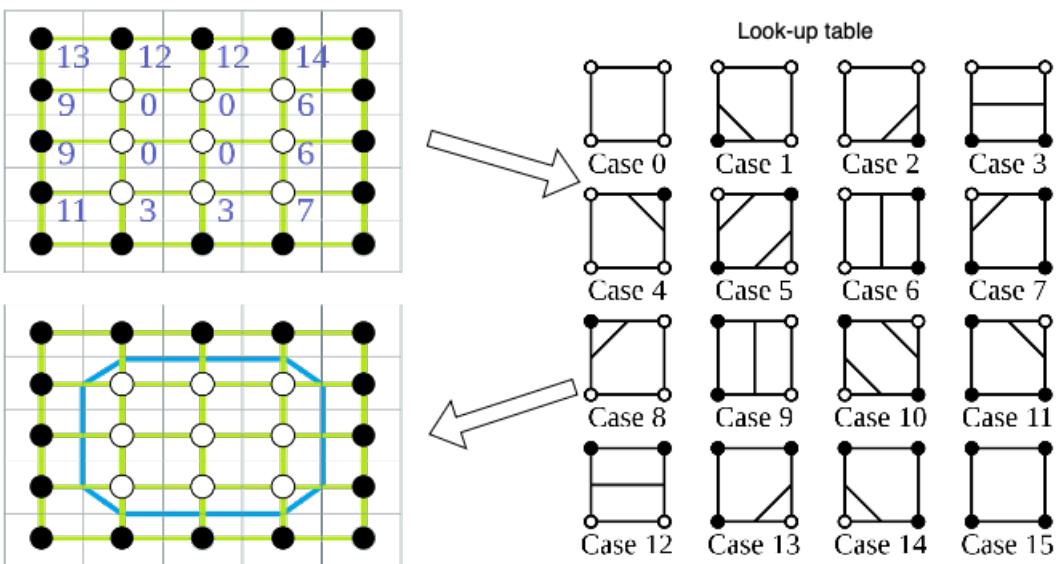


Figura 1.2: Schema funzionamento dell’individuazione di contorni in Marching Squares.

Nella figura 1.3 è possibile vedere una generica rappresentazione a colori dei sedici casi possibili nella versione più comune di Marching Squares, equivalente a quella riportata in bianco e nero nella figura 1.2. 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 limiti superiori e inferiori come rappresentato nell’immagine 1.4.

Esistono anche versioni che invece dei quadrati utilizzano triangoli e vengono applicate per l’individuazione di meshes triangolari.

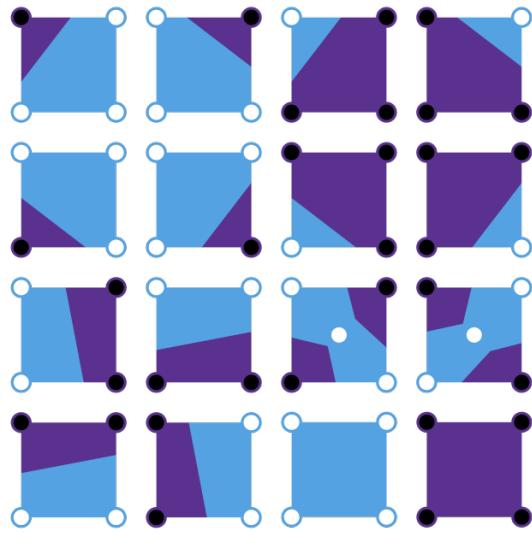


Figura 1.3: Sedici casi possibili in cui possono ricadere i quadrati composti dai quattro valori.

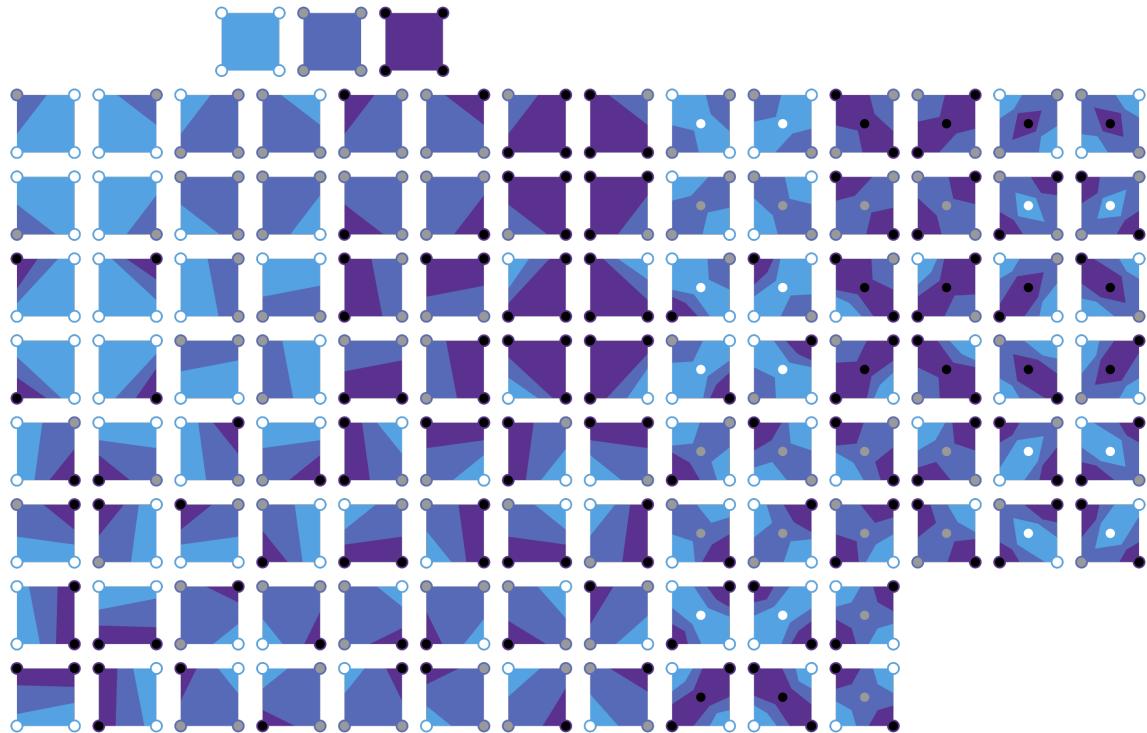


Figura 1.4: Casi possibili nella versione con isobande in cui possono ricadere i quadrati composti dai quattro valori.

Per una più comprensione più accurata della logica sottostante all'algoritmo MS è utile osservare il suo pseudocodice che è riportato in seguito.

Il metodo principale è `_get_contour_segments` dove la parte più importante dell'algoritmo è implementata, è stato riportato per completezza anche il metodo `_get_fraction` dato che viene richiamato nella costruzione delle coordinate (righe 30-33).

---

## Pseudocodice Marching Squares

---

```
1 cdef _get_fraction( cnp.float64_t from_value,
2                     cnp.float64_t to_value,
3                     cnp.float64_t level):
4     if (to_value == from_value):
5         return 0
6     return ((level - from_value) / (to_value - from_value))
7
8 def _get_contour_segments(cnp.float64_t [:, :] array,
9                           cnp.float64_t level):
10    list segments = []
11
12   for r0 in range(array.shape[0] - 1):
13       for c0 in range(array.shape[1] - 1):
14           r1, c1 = r0 + 1, c0 + 1
15
16           ul = array[r0, c0]
17           ur = array[r0, c1]
18           ll = array[r1, c0]
19           lr = array[r1, c1]
20
21           square_case = 0
22           if (ul > level): square_case += 1
23           if (ur > level): square_case += 2
24           if (ll > level): square_case += 4
25           if (lr > level): square_case += 8
26
27           if square_case in [0, 15]:
28               continue
29
30           top = r0, c0 + _get_fraction(ul, ur, level)
31           bottom = r1, c0 + _get_fraction(ll, lr, level)
32           left = r0 + _get_fraction(ul, ll, level), c0
33           right = r0 + _get_fraction(ur, lr, level), c1
34
35           if (square_case == 1):
36               segments.append((top, left))
37           elif (square_case == 2):
38               segments.append((right, top))
```

```
39     elif (square_case == 3):
40         segments.append((right, left))
41     elif (square_case == 4):
42         segments.append((left, bottom))
43     elif (square_case == 5):
44         segments.append((top, bottom))
45     elif (square_case == 6):
46         segments.append((left, top))
47         segments.append((right, bottom))
48     elif (square_case == 7):
49         segments.append((right, bottom))
50     elif (square_case == 8):
51         segments.append((bottom, right))
52     elif (square_case == 9):
53         segments.append((top, right))
54         segments.append((bottom, left))
55     elif (square_case == 10):
56         segments.append((bottom, top))
57     elif (square_case == 11):
58         segments.append((bottom, left))
59     elif (square_case == 12):
60         segments.append((left, right))
61     elif (square_case == 13):
62         segments.append((top, right))
63     elif (square_case == 14):
64         segments.append((left, top))
65 return segments
```

---

L'algoritmo è embarrassingly parallel per quanto riguarda la classificazione per tipo di ogni cella (quadrato con valori negli spigoli) poiché può essere svolta in modo indipendente tra le celle.

La fase di ricostruzione dei contorni invece può essere svolta sia in parallelo che serialmente utilizzando tecniche e modalità differenti che dipendono dall'utilizzo finale a cui il codice è 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 identificate 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.

### 1.1.2 Problema Reale

Bioretics è l'azienda con cui è stata svolta la ricerca, opera in diversi settori tra i quali la 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 (Convolutional Neural Network) 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, per cui c'è approssimativamente 0.1s a disposizione per ogni frutto. 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  $W \times H \times C$ , 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 1.5 dove si possono notare le varie classi delimitate da colori differenti.



Figura 1.5: 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.

### 1.1.3 Obbiettivo

L'obbiettivo concordato con l'azienda è quello di implementare una versione parallela su GPU (scheda video) dell'algoritmo Marching Squares. Ottenere di uno speedup rispetto alla versione utilizzata attualmente ovvero il metodo `find_contours` della libreria skimage sarebbe considerata un successo.

Il software dell'azienda che attualmente richiama la funzione `find_contours` 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 1.6 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 interfacerà il codice dell'azienda, sarà quindi un modulo Python da cui poi si cercherà una strategia per arrivare all'esecuzione su GPU.

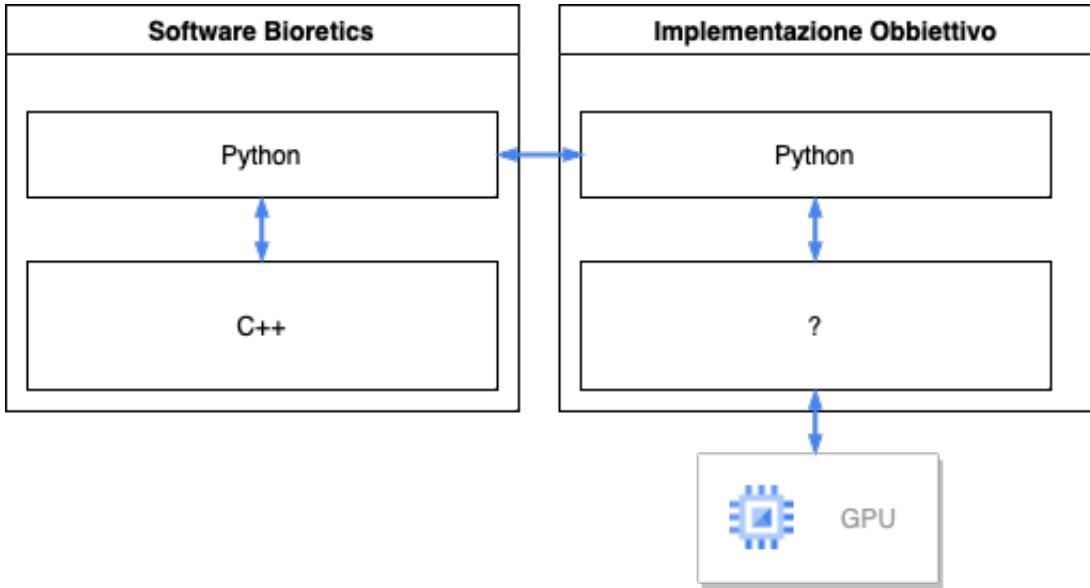


Figura 1.6: Schema linguaggi utilizzati dal software dell’azienda e da utilizzate per la soluzione finale.

## 1.2 Analisi implementazione skimage

Il metodo `find_contours` di skimage [5] 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 nella seconda 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 [5]. 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 al massimo uguale ma 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, allora il codice che ne deriva risulta estremamente più veloce della versione Python, con tempi di esecuzione si avvi-

cinano alla versione C che teoricamente può essere considerata un suo limite inferiore. Queste osservazioni sono riscontrabili graficamente nei risultati dei test generici riportati nella figura 1.7.

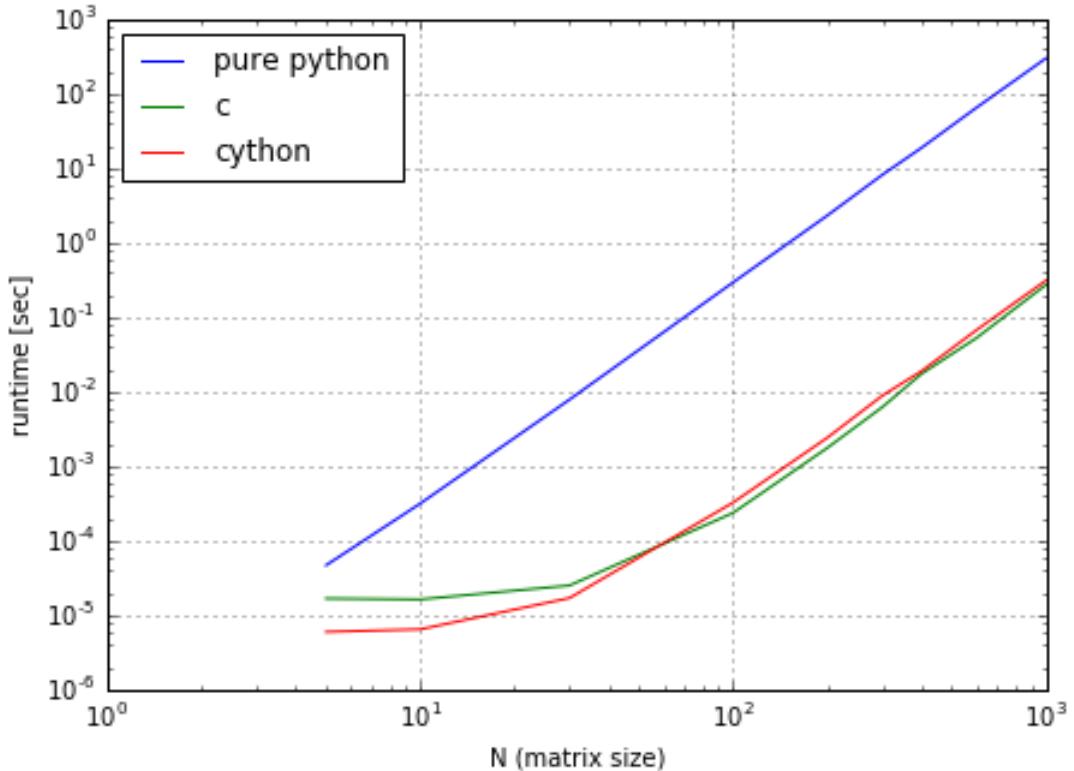


Figura 1.7: Grafico che mette in comparazione i tempi di esecuzione di Python, C e Cython su un task di esempio.

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 reale di  $511 \times 95$  il metodo `findContours` impiega in media circa 0.001192s ovvero neanche tre millesimi di secondo.

## 1.3 Introduzione architettura GPU

Una scheda video o GPU (Graphics Processing Unit) è un'unità di elaborazione progettata per l'elaborazione parallela. Viene solitamente sfruttata per l'elaborazione grafica ma è ampiamente utilizzata anche per applicazioni di calcolo intensivo dove è possibile parallelizzare le operazioni da eseguire.

Nella terminologia CUDA la GPU viene chiamata **device**. CUDA definisce infatti il device come l'insieme di GPU (intesa come unità di calcolo) e delle sue memorie. Ci si riferisce invece come **host** alla CPU e le sue memorie.

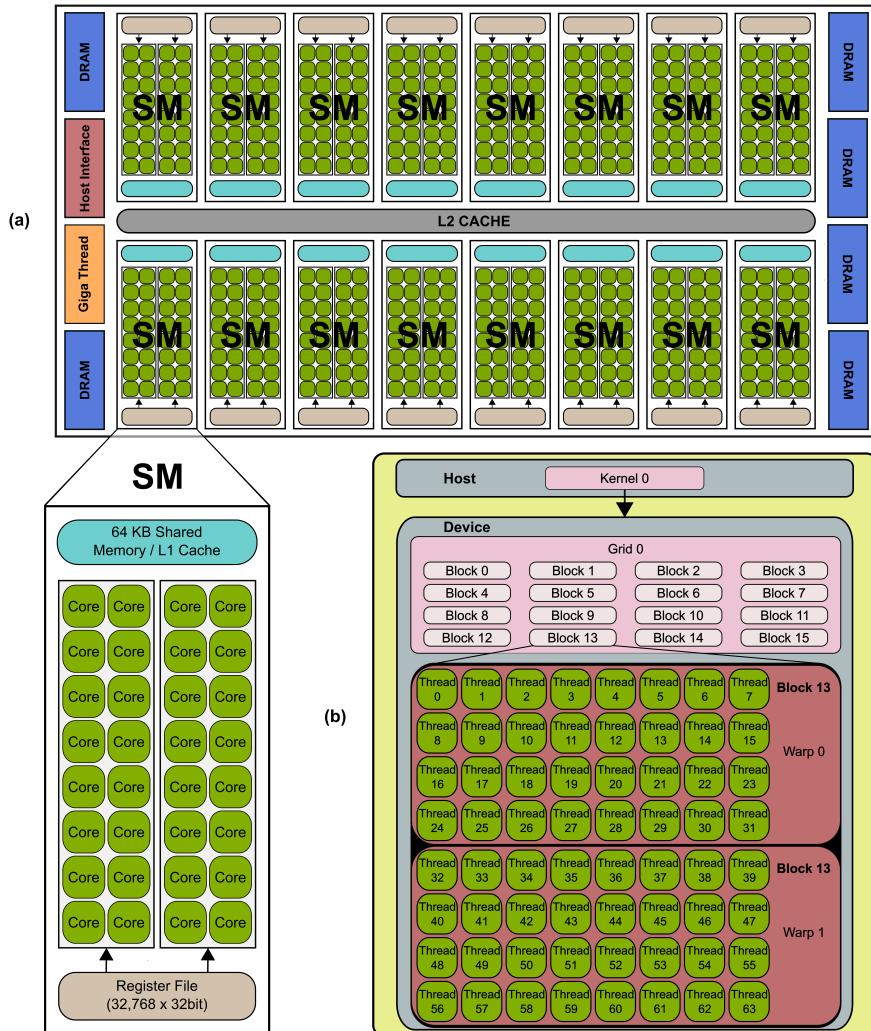


Figura 1.8: Rappresentazione dell'architettura di un GPU.

Come è possibile osservare dalla figura 1.8 l'hardware di una GPU è composto principalmente da unità logiche e memorie di vario genere. In primo piano si ha la memoria globale, l'interfaccia con l'host e degli Streaming Multiprocessor. Approfondendo, all'interno di ogni Streaming Multiprocessor sono presenti gli Streaming Processors, delle memorie cache e una memoria condivisa.

L'unità di lavoro minima su una GPU è il **thread**. Quando viene lanciato un kernel, le sue istruzioni vengono eseguite da tutti i thread, che sono raggruppati in **warp** ovvero insiemi di 32 thread per i quali è garantita l'esecuzione in parallelo. Questo implica che il minimo numero di thread utilizzabili per un'operazione è 32, che la eseguiranno in parallelo sfruttando il paradigma SIMD (Single Instruction stream, Multiple Data stream).

I thread vengono raggruppati in array 3D chiamati **block** che sono delle porzioni di lavoro indipendente che possono essere eseguiti da uno Streaming Multiprocessor in qualsiasi ordine. I blocchi vengono a loro volta raggruppati in array 2D (o 3D) chiamati **grid** che sono pezzi di lavoro assegnabile ad una GPU.

## 1.4 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 colli di bottiglia (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  $511 \times 95$ ; con una versione parallela del Marching Squares idealmente ogni thread si occuperebbe di un singolo quadrato, con immagini di queste dimensioni servirebbero 47940 thread (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 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 collo di bottiglia 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.

## 1.5 Analisi versione seriale di MS

La libreria **scikit-image** (skimage) è un pilastro per quanto riguarda l'elaborazione di immagini in ambito Open-Source essendo largamente utilizzata sia in piccoli progetti che in contesti industriali. Il codice delle funzioni più utilizzate è solido e ottimizzato pur mantenendo un'ampia compatibilità con una vasta lista di ambienti. Tra queste funzioni ricade anche **find\_contours** che infatti presenta buone prestazioni per essere una funzione seriale lanciata da Python.

Dal codice di **find\_contours** che è riportato nella sezione di codice seguente, 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.

---

### 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 del metodo Cython.

---

### Estratto dal Metodo `_get_contour_segments` di scikit-image

---

```
1 cimport numpy as np
2 np import_array()
3 cdef extern from "numpy/npy_math.h":
4     bint npy_isnan(np.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 np.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 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 di MS numerati come 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 1.9.

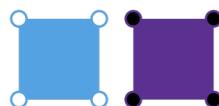


Figura 1.9: Caso 0 e 15 dell'algoritmo Marching Squares.

- 2 segmenti da aggiungere: per i tipi di MS numerati come 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 1.10.



Figura 1.10:  
Caso 6 e 9  
dell'algoritmo  
Marching Squa-  
res.

- 1 segmento da aggiungere: tutti gli altri tipi di MS esclusi quelli numerati 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 1.11.

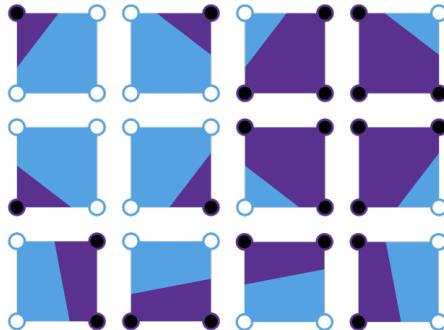


Figura 1.11: Casi 1-5,7,8,10-14  
dell'algoritmo Marching Squares.

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` viene riportata l'assegnazione delle coordinate per i punti che potranno essere aggiunti alla lista come estremi dei segmenti individuati.

---

### Estratto dal Metodo `_get_contour_segments` di scikit-image

---

```
1 cdef inline cnp.float64_t _get_fraction(cnp.float64_t from_value,
2                                         cnp.float64_t to_value,
3                                         cnp.float64_t level):
4     if (to_value == from_value):
5         return 0
6     return ((level - from_value) / (to_value - from_value))
7
8 top = r0, c0 + _get_fraction(ur, lr, level)
9 bottom = r1, c0 + _get_fraction(ll, lr, level)
10 left = r0 + _get_fraction(ur, ll, level), c0
11 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

---

```
1 # Manage case 0 and 15
2 if square_case in [0, 15]:
3     continue
4
5 # Example for case 1-5, 7, 8, 10-14
6 if (square_case == 1):
7     # top to left
8     segments.append((top, left))
9 # Example for case 6 and 9
10 elif (square_case == 6):
11     segments.append((left, top))
12     segments.append((right, bottom))
```

---

### 1.5.1 Predisposizione alla parallelizzazione

L'algoritmo Marching Squares essendo embarrassingly 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 poiché 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 2

## Versione parallela con nvc++

La prima strategia esplorata consiste nel compilare il codice Cython con le ultime versioni del compilatore nvc++ che supporta dei flag per la parallelizzazione automatica su GPU. Nvidia ha recentemente aggiornato `stdpar`, un flag che promette di parallelizzare automaticamente codice C++ su GPU utilizzando il compilatore nvc++ [2].

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 su GPU tra cui `stdpar`; infine si può richiamare il codice parallelizzato da Python tramite l'importazione del modulo generato.

Il processo documentato da Nvidia è descritto in forma schematica nella figura 2.1.

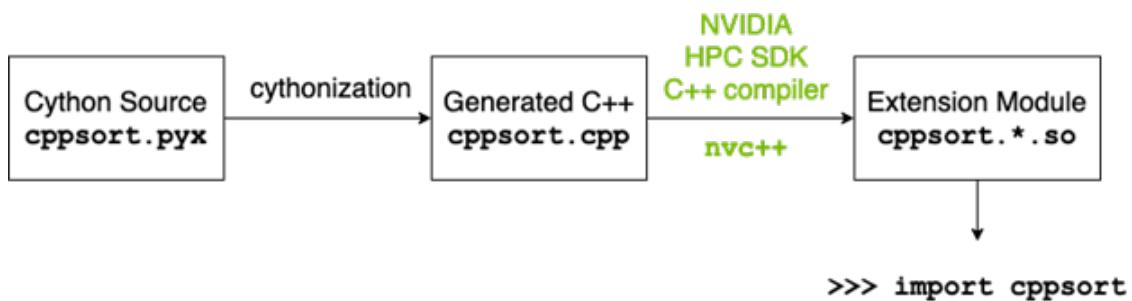


Figura 2.1: Schema di descrizione processo per utilizzo nvc++ in parallelizzazione codice Cython prodotto a Nvidia.

## 2.1 Requisiti utilizzo nvc++

Per poter utilizzare il flag stdpar con una versione aggiornata di nvc++ è necessario installare *Nvidia High Performance Computing Software Development Kit (HPC SDK)* sulla macchina su cui verrà effettuata la compilazione e l'esecuzione. Le opzioni disponibili per ottenere HPC SDK sono installarlo direttamente sulla macchina oppure utilizzare un container Docker configurato da Nvidia con tutte le dipendenze di nvc++ installate.

L'azienda Bioretics ha messo a disposizione un server con una scheda video RTX 2060 Super. Per evitare tutte le problematiche legate alle dipendenze come le loro versioni e il loro mantenimento è stato valutato di utilizzare la soluzione con Docker.

L'utilizzo del container Docker fornito da Nvidia si è rivelato però particolarmente ostico poiché per il suo avvio è stata necessaria una configurazione molto articolata che non era descritta nella guida all'utilizzo. Per il corretto funzionamento del container e del compilatore nvc++ è stata necessaria una particolare configurazione che permetesse al container di utilizzare la scheda video e i suoi driver con permessi specifici. L'ambiente ottenuto sul server con la configurazione del container è rappresentato dalla figura 2.2.

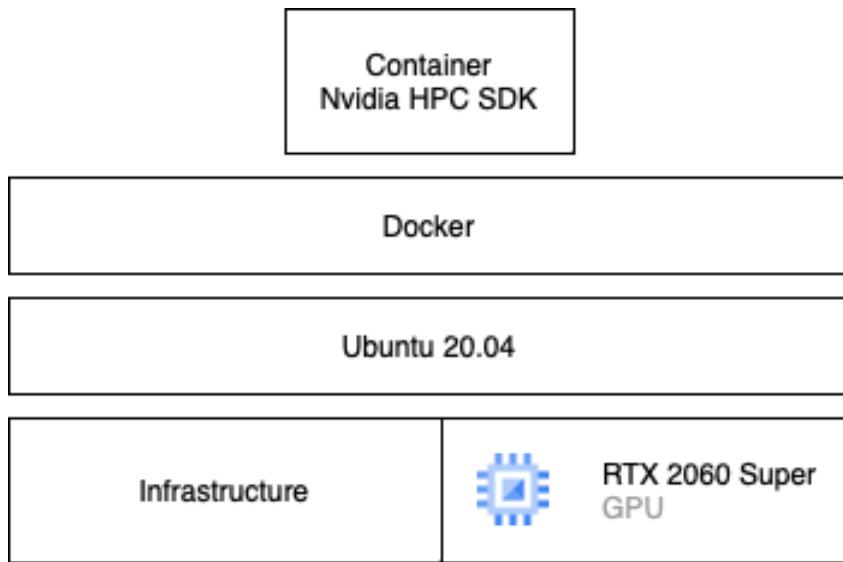


Figura 2.2: Schema componenti hardware e software sul server dell'azienda in seguito alla configurazione del container Nvidia HPC SDK.

## 2.2 Potenziali risultati

I risultati promessi da Nvidia con l'utilizzo di nvc++ e `-stdpar` sembrano avere un alto potenziale ma vanno ben contestualizzati ai task eseguiti e all'hardware utilizzato.

I benchmark effettuati dal team di sviluppo Nvidia sono sull'ordinamento di una serie di numeri e su iterazioni del metodo di Jacobi; entrambe sono due funzioni facilmente parallelizzabili, largamente studiate ed estremamente ottimizzate nelle implementazioni delle principali librerie. Per ognuno dei due task relativi alle due funzioni appena descritte sono state utilizzate 3 versioni:

- Versione che utilizza la funzione seriale.
- Versione parallela su CPU che utilizza le policy di esecuzione parallela ed è compilata con g++.
- Versione parallela su GPU che utilizza sempre le policy di esecuzione parallela ma è compilata con nvc++ e l'opzione `-stdpar`.

Nei test effettuati da Nvidia sull'ordinamento di numeri presentati in figura 2.3 è rappresentato lo speedup delle tre versioni rispetto all'implementazione di Numpy della funzione `sort()`. Si può notare uno speedup di circa 20 volte nel test con il numero maggiore di elementi per la versione parallelizzata su GPU da nvc++ con `-stdpar`; risultato apparentemente molto promettente. Nei test con un basso numero di elementi risultano però migliori le due versioni su CPU (seriali e parallele). Questo fenomeno è dovuto all'overhead introdotto da un'esecuzione parallela su GPU che comporta il trasferimento di dati sulle memorie. Il problema è lo stesso discusso precedentemente in relazione alle dimensioni ridotte delle immagini utilizzate nell'applicazione reale oggetto di questa ricerca.

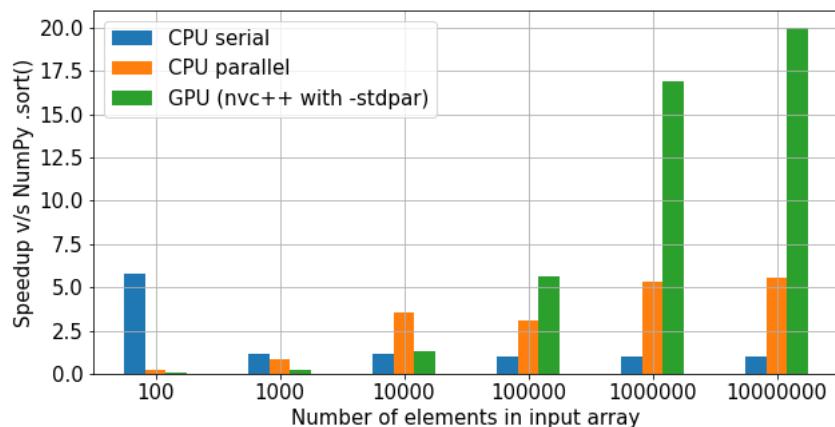


Figura 2.3: Speedup ottenuto a confronto con Numpy nell'ordinamento di una sequenza di interi. I benchmark su GPU sono stati eseguiti su un sistema con Intel Xeon Gold 6128 CPU, quelli su GPU invece su una NVIDIA A100.

Nei test sulle iterazione del metodo di Jacobi il fenomeno riscontrato con `.sort()` non si presenta come osservabile nella figura 2.4. Il test con il numero di elementi minore comporta un carico di lavoro nettamente maggiore rispetto a quello del test precedente ed è quindi già in grado di ammortizzare il costo impiegato per l'esecuzione parallela su GPU. I risultati ottenuti in questi benchmark sono ancora più eclatanti, tanto da insospettirsi sul fatto che siano realmente ottenibili anche su altre funzioni di utilizzo reale.

Bisogna considerare infatti che tutti i risultati ottenuti in questi test con il compilatore nvc++ e la direttiva `-stdpar` sono stati eseguiti su una macchina con componenti hardware di fascia altissima che sviluppano una potenza di calcolo estremamente lontana da quella della maggior parte delle macchine utilizzate in ambito enterprise. Infatti la GPU utilizzata da Nvidia per questi test è una Nvidia A100 ovvero una scheda che si trova in configurazioni con decine di GB di memoria a costi molto maggiori a quelli di una generica GPU di fascia alta.

Oltre all'hardware bisogna anche considerare che i task su cui sono stati effettuati i test comprendevano esclusivamente esecuzione di codice interamente parallelizzabile; strutturalmente quindi molto distante dal codice utilizzato come base per questo progetto.

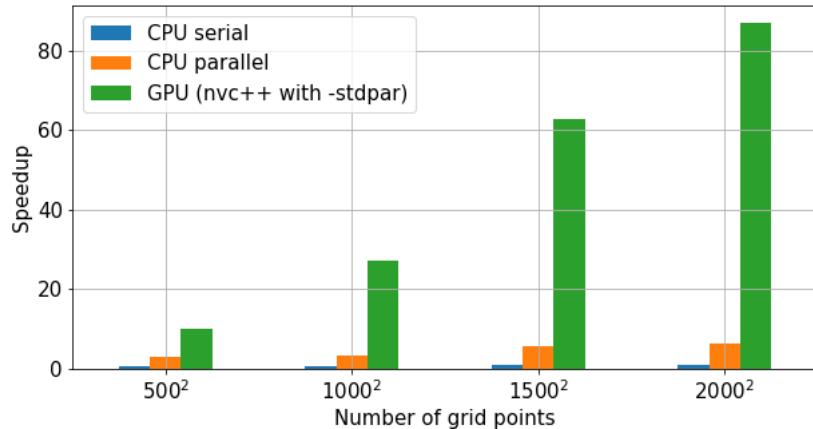


Figura 2.4: Speedup ottenuto a confronto con Numpy in iterazione di Jacobiano. I benchmark su GPU sono stati eseguiti su un sistema con Intel Xeon Gold 6128 CPU, quelli su GPU invece su una NVIDIA A100.

## 2.3 Marching Squares con nvc++ e -stdpar

Il risultato migliore che potrebbe essere raggiunto è quello di riuscire a ottenere anche solo una frazione dello speedup riportato da Nvidia con nvc++ e -stdpar utilizzandoli con il codice Cython.

La strategia scelta per arrivare a questo risultato è quella di partire da una versione di codice basilare e poi evolverla in quella finale per raffinamenti successivi, in modo da risolvere in modo incrementale i problemi che inevitabilmente si presentano in implementazioni di questo tipo dove sono compresi linguaggi diversi e compilatori complessi.

I risultati ottenuti dal team di sviluppo di Nvidia si sono subito rivelati un lontano traguardo da raggiungere con hardware di fascia media. Testando infatti lo stesso codice con cui erano stati effettuati i benchmark, sono stati ottenuti risultati che non erano neanche simili. La versione parallela per GPU generata automaticamente dal compilatore è risultata molto meno efficiente delle versioni seriali e parallele su CPU. Ovviamente risulta difficile ottenere certi risultati sulla macchina su cui poi verrà utilizzato il codice con il proprio progetto se neanche il codice fornito da Nvidia (perfettamente ottimizzato) riesce a raggiungerli. Dato che per il successo del progetto però è sufficiente riuscire a parallelizzare su GPU anche solo una piccola parte del codice e ottenere uno speedup, questa strada è stata considerata ancora valida.

Il codice proposto da Nvidia era particolarmente semplice e da poche righe totali, la compilazione aveva quindi poche pretese. Lo stesso non vale per il codice Cython di skimage che comprende importazioni da librerie esterne e il loro utilizzo. Nel file `setup.py` sono specificate tutte le direttive per la compilazione con nvc++ e viene gestito in collegamento di tutte le librerie necessarie. Per concludere senza errori la compilazione del codice di skimage è stato quindi necessario modificare la struttura e il contenuto delle opzioni di compilazione nel file `setup.py` che viene passato a nvc++ per definire tutte le opzioni di compilazione e le librerie da includere.

Il processo parte dalla trasformazione del codice Cython in C++ con `cythonize`.

---

### Cythonizzazione

---

```
1 $ cythonize -i _find_contours_cy.pyx
```

---

Il codice C++ ottenuto viene poi compilato con il seguente comando.

---

### Compilazione con nvc++ e -stdpar

---

```
1 $ CC=nvc++ python setup.py build_ext --inplace
```

---

Il risultato di questo processo è un modulo importabile da Python utilizzabile per richiamare

il metodo che verrà poi eseguito in parallelo su GPU.

## 2.4 Problemi riscontrati

Purtroppo tutti i tentativi effettuati non hanno portato ad una versione parallela funzionante su GPU. Il principale problema riscontrato è la complessità e disordine del codice C++ derivato dal processo di cythonizzazione del codice Cython: il codice in output ha infatti dimensioni di ordini di grandezza maggiori. Da un centinaio di righe in Cython il codice C++ risultante passa a decine di migliaia che risultano quindi ingestibili dato che ci andrebbero aggiunte manualmente le policy di esecuzione parallela per ogni singolo metodo della libreria standard che si vuole parallelizzare su GPU. Nei test effettuati, anche inserendo le policy per la parallelizzazione, l'esecuzione fallisce a causa delle strutture dati generate dalla cythonizzazione come puntatori a strutture che non sono riconosciute nativamente da C++ e quindi considerate come un insieme di byte. La parallelizzazione automatica su GPU di nvc++ con `-stdpar` sul codice C++ derivato dal Cython si è rivelata quindi estremamente difficile da utilizzare con codice più complesso di quello proposto da Nvidia che comprendeva praticamente solo qualche funzione base della libreria standard C++. In seguito a queste conclusioni è stato valutato di non proseguire con questa strada ma intraprenderne una più complessa in termini di implementazione su cui però si può avere più controllo nel processo di parallelizzazione su GPU (descritto nel capitolo 5).

# Capitolo 3

## Versione parallela con API CUDA-Python

### 3.1 API Cuda-Python

Cuda-Python è un progetto nato per unificare l'ecosistema di utilizzo di CUDA da Python con un insieme di interfacce a basso livello in grado di mettere a disposizione una copertura completa all'accesso da codice Python alle API dell'host CUDA [4]. Il principale obiettivo è quello di agevolare i programmatore Python nell'utilizzo di GPU Nvidia.

Per poter utilizzare le API Cuda-Python è necessario il CUDA Toolkit (dalla versione 12.0 alla 12.2) e predisporre una macchina che abbia una versione aggiornata sia di Python (da 1.8 a 1.11) che dei Driver Nvidia specifici per le proprie schede video.

Dal CUDA Toolkit è richiesto solamente il componente NVRTC che è utilizzato a runtime per la compilazione di codice CUDA C++.

La scheda video della macchina su cui viene utilizzato deve essere Nvidia e compatibile con l'utilizzo di CUDA.

#### 3.1.1 CUDA Python workflow

Il codice scritto in Cuda deve essere compilato per poter esser eseguito su una GPU. Python invece è un linguaggio interpretato che non necessita di una previa compilazione ma viene eseguito riga per riga. Per riuscire a richiamare un metodo Cuda da Python è necessario compilare il codice del device (codice Cuda che verrà eseguito su GPU) in PTX (Parallel Thread Execution) ed estrarne la funzione che potrà poi essere richiamata dall'applicazione python. PTX è una macchina virtuale per l'esecuzione di thread paralleli a basso livello che comprende un ISA (Instruction Set Architecture) ossia un insieme di istruzioni macchina.

In pratica il codice del device viene scritto in una stringa python e compilato con NVRTC che è una libreria di compilazione runtime per CUDA C++.

In generale il processo di esecuzione parallela del codice Cuda partendo da codice Python consiste nell'utilizzo delle API dei Driver Nvidia per:

- Creazione manuale del `context` CUDA su GPU
- Allocazione manuale di tutte le risorse necessarie su GPU
- Compilazione codice Cuda (contenuto in stringa Python) con NVRTC
  - Nella seguente figura 3.1 viene riportato uno schema su questo processo di compilazione

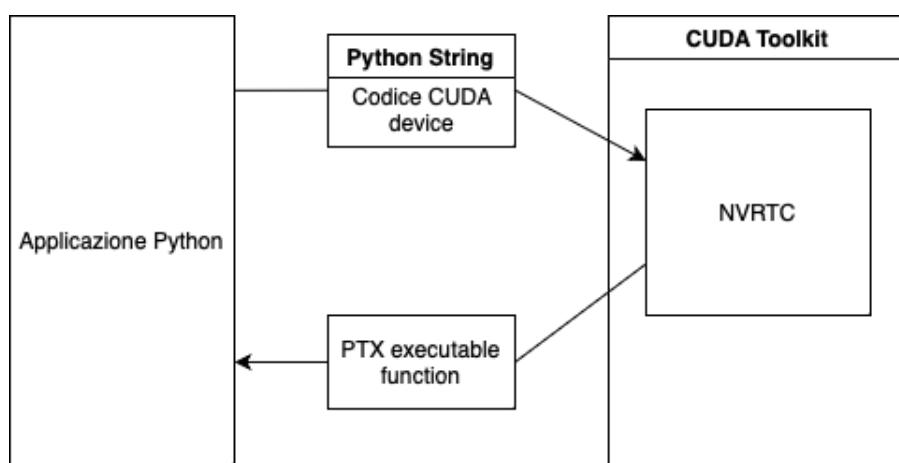


Figura 3.1: Schema compilazione runtime con NVRTC di codice CUDA C++ da codice Python e output funzione compatibile PTX richiamabile da Python.

- Caricamento dati da elaborare su GPU
- Lancio codice CUDA C++ compilato su GPU
- Recupero risultati da GPU

### 3.1.2 Utilizzo funzioni principali

Per prima cosa vanno importati NVRTC e le API dei Driver dal pacchetto CUDA-Python come riportato nella seguente porzione di codice. Per copiare dati dall'host al device e viceversa è richiesto l'utilizzo di NumPy.

---

#### Importazione NVRTC, Driver API e NumPy

---

```
1 from cuda import cuda, nvrtc
2 import numpy as np
```

---

In seguito viene creato il programma a partire dal codice CUDA che può essere mantenuto in una stringa o portato in un file con estensione ".cu" per mantenere ordine e chiarezza. Il programma deve poi essere compilato e dal risultato ne deve essere estratta la funzione PTX che potrà essere richiamata dal Python. In questo codice di esempio viene cercata una capacità di computazione di 75.

---

#### Creazione, compilazione e estrazione funzione PTX da codice CUDA C++

---

```
1 # Create program
2 err, prog = nvrtc.nvrtcCreateProgram(      str.encode(saxpy),
3                                         b"saxpy.cu",
4                                         0, [], []    )
5
6 # Compile program
7 opts = [b"--fmad=false", b"--gpu-architecture=compute_75"]
8 err, = nvrtc.nvrtcCompileProgram(prog, 2, opts)
9
10 # Get PTX from compilation
11 err, ptxSize = nvrtc.nvrtcGetPTXSize(prog)
12 ptx = b" " * ptxSize
13 err, = nvrtc.nvrtcGetPTX(prog, ptx)
```

---

Prima di poter utilizzare il PTX o eseguire qualsiasi cosa sulla GPU è necessario creare un CUDA context, che è l'equivalente di un processo per la CPU. Successivamente è necessario inizializzare le API dei Driver per far in modo che i driver Nvidia e la GPU siano accessibili. Per assegnare la GPU principale della macchina va passato l'indicatore 0 alla funzione `cuCtxCreate` per la creazione del `context`. Con il `context` creato si può procedere alla compilazione del CUDA kernel con NVRTC.

---

## Inizializzazione Driver API e creazione context su device 0

---

```
1 # Initialize CUDA Driver API
2 err, = cuda.cuInit(0)
3
4 # Retrieve handle for device 0
5 err, cuDevice = cuda.cuDeviceGet(0)
6
7 # Create context
8 err, context = cuda.cuCtxCreate(0, cuDevice)
```

---

Con un CUDA `context` creato sul device 0 si può caricare nel modulo il PTX generato in precedenza. Per il device un modulo è l'analogo di una libreria caricata dinamicamente. Più kernel posso essere contenuti all'interno di PTX; in seguito al caricamento nel modulo è possibile estrarre uno specifico kernel con la funzione `cuModuleGetFunction`.

---

## Caricamento PTX come modulo ed estrazione funzione

---

```
1 # Get PTX
2 ptx = np.char.array(ptx)
3
4 # Load PTX as module data
5 err, module = cuda.cuModuleLoadData(ptx.ctypes.data)
6 ASSERTDRV(err)
7
8 # Retrieve function
9 err, kernel = cuda.cuModuleGetFunction(module, b"saxpy")
10 ASSERTDRV(err)
```

---

Tutti i dati a cui sarà necessario accedere dai kernel devono essere preparati e trasferiti sulla GPU; prima è necessario però allocare le risorse necessarie a contenere i dati utilizzando la funzione `cuMemAlloc`.

Una delle principali differenze tra Python e CUDA C è l'utilizzo di strutture dati differenti sia come tipo che livello di astrazione, Python nativamente non ha un concetto di puntatore ma la funzione `cuMemcpyHtoDAsync` si aspetta `void*` ovvero un puntatore senza un tipo specificato. La soluzione introdotta da Nvidia è quella di usare `XX.ctypes.data` che recupera il valore del puntatore associato a `XX`.

---

## Creazione, allocazione e trasferimento dati su GPU

---

```
1 NUMTHREADS = 512 # Threads per block
2 NUMBLOCKS = 32768 # Blocks per grid
3
4 a = np.array([2.0], dtype=np.float32)
5 n = np.array(NUMTHREADS * NUMBLOCKS, dtype=np.uint32)
6 bufferSize = n * a.itemsize
7
8 hIn = np.random.rand(n).astype(dtype=np.float32)
9 hOut = np.zeros(n).astype(dtype=np.float32)
10
11 err, dInclass = cuda.cuMemAlloc(bufferSize)
12 err, dOutclass = cuda.cuMemAlloc(bufferSize)
13
14 err, stream = cuda.cuStreamCreate(0)
15
16 err, = cuda.cuMemcpyHtoDAsync(
17     dInclass, hIn.ctypes.data, bufferSize, stream
18 )
```

---

Con i dati caricati i kernel sono pronti per essere lanciati; per farlo è necessario avere a disposizione gli indirizzi dei dati sul device in modo da passarli come parametri ai kernel. Per ottenere questi puntatori è possibile utilizzare `int(dData)` che restituisce un tipo `CUdeviceptr` e assegna un dimensione in termini di memoria con cui può essere memorizzato il valore utilizzando `np.array()`.

---

## Recupero puntatori e allocazione parametri per funzione kernel

---

```
1 dIn = np.array([int(dInclass)], dtype=np.uint64)
2 dOut = np.array([int(dOutclass)], dtype=np.uint64)
3
4 args = [a, dIn, dOut, n]
5 args = np.array([arg.ctypes.data for arg in args], dtype=np.uint64)
```

---

Il kernel può essere lanciato con la funzione `cuLaunchKernel` a cui vanno passati il modulo compilato del kernel e i parametri di configurazione per l'esecuzione. Lo stream creato in precedenza viene utilizzato sia per il trasferimento dei dati che per il lancio del codice del device; in questo modo il kernel inizia la sua esecuzione solamente quando è terminato il trasferimento di tutti i dati. Tutte le chiamate alle API e i lanci dei kernel su uno stream sono quindi serializzati.

---

## Lancio kernel

---

```
1 err, = cuda.cuLaunchKernel(
2     kernel,
3     NUM_BLOCKS, # grid x dim
4     1, # grid y dim
5     1, # grid z dim
6     NUM_THREADS, # block x dim
7     1, # block y dim
8     1, # block z dim
9     0, # dynamic shared memory
10    stream, # stream
11    args.ctypes.data, # kernel arguments
12    0, # extra (ignore)
13 )
```

---

Dato che le chiamate alle API sono eseguite in sequenza si può richiedere la copia del risultato dal device all'host direttamente dopo il lancio del kernel: la chiamata infatti verrà risolta solo al termine dell'esecuzione del kernel.

Prima di proseguire con altre chiamate (per esempio a nuovi kernel) è utile richiamare la funzione `cuStreamSynchronize` che mette in attesa la CPU fino a quando tutte le operazioni dello stream corrente non sono terminate.

---

## Trasferimento risultato da Device (GPU) a Host

---

```
1 err , = cuda . cuMemcpyDtoHAsync(            
2     hOut . ctypes . data , dOutclass , bufferSize , stream  
3 )  
4 err , = cuda . cuStreamSynchronize ( stream )
```

---

Come ultima operazione è opportuno liberare le risorse allocate effettuando una pulizia della memoria allocata su GPU e distruggendo sia lo **stream** che il **context** creati.

---

## Liberazione risorse

---

```
1 err , = cuda . cuStreamDestroy ( stream )  
2 err , = cuda . cuMemFree ( dInclass )  
3 err , = cuda . cuMemFree ( dOutclass )  
4 err , = cuda . cuModuleUnload ( module )  
5 err , = cuda . cuCtxDestroy ( context )
```

---

Con queste ultime funzioni di pulizia si conclude l'introduzione ai comandi messi a disposizione delle API CUDA-Python che possono essere utilizzati per lanciare codice CUDA su GPU da codice Python.

La versione di CUDA-Python utilizzata (12.2.0) è ancora in evoluzione e potrebbe quindi essere aggiornata in futuro rendendo le funzioni precedenti o procedure descritte deprecate; è consigliato quindi consultare la pagina ufficiale di Nvidia per una documentazione aggiornata.

### 3.1.3 Prestazioni dichiarate

Nvidia riporta i benchmark effettuati su del codice di base, ovvero tutte le istruzioni descritte in precedenza per l'utilizzo delle API con un semplice kernel CUDA che effettua operazioni di algebra lineare chiamato SAXPY (Single-Precision A·X Plus Y).

Ogni thread effettua solamente una operazione  $a \cdot x + y$  utilizzando un valore scalare e due valori letti da una coppia di array in input, scrive poi il risultato su un array di output. La semplice implementazione CUDA utilizzata per i benchmark è la seguente.

---

#### Codice CUDA per device utilizzato nei benchmark

---

```
1 saxpy = """\
2 extern "C" __global__
3 void saxpy(float a, float *x, float *y, float *out, size_t n)
4 {
5     size_t tid = blockIdx.x * blockDim.x + threadIdx.x;
6     if (tid < n) {
7         out[tid] = a * x[tid] + y[tid];
8     }
9 }
10 """
```

---

I risultati ottenuti dal team di Nvidia sono riportati nella tabella 3.1 in cui si può notare come la versione C++ e Python appaiano estremamente simili come tempi di esecuzione, sia per le pure esecuzioni dei kernel che per il tempo totale dell'applicazione. Si può osservare come la versione Python introduca un piccolo overhead riscontrabile nella differenza tra i due tempi di esecuzione dell'applicazione intera che differiscono di qualche millisecondo. Le versioni kernel invece non hanno alcun tipo di differenza nei tempi di esecuzione, questo significa che una volta che i kernel sono lanciati la loro esecuzione non viene influenzata dal metodo utilizzato per lanciarli.

	C++	Python
Kernel execution	352 µs	352 µs
Application execution	1076 ms	1080 ms

Tabella 3.1: Comparazione tempi di esecuzione kernel e applicazione completa tra versione in C++ e Python.

## 3.2 Progettazione struttura codice parallelo

La soluzione da sviluppare deve essere in grado di usufruire delle risorse GPU per i calcoli dell'algoritmo Marching Squares partendo da una chiamata ad un metodo Python, per poter essere integrato al codice utilizzato da Bioretics.

L'idea che comprende l'utilizzo delle API CUDA-Python è di creare un modulo Python richiamabile dal codice dell'azienda che tramite le API dei Driver CUDA riesca a caricare i dati, lanciare i kernel e recuperare il risultato dalla GPU per poi restituirlo all'applicazione Python. Può essere quindi definito il nuovo quadro generale per questa soluzione che definisce il ponte tra il codice Python e il codice CUDA osservabile nella figura 3.2.

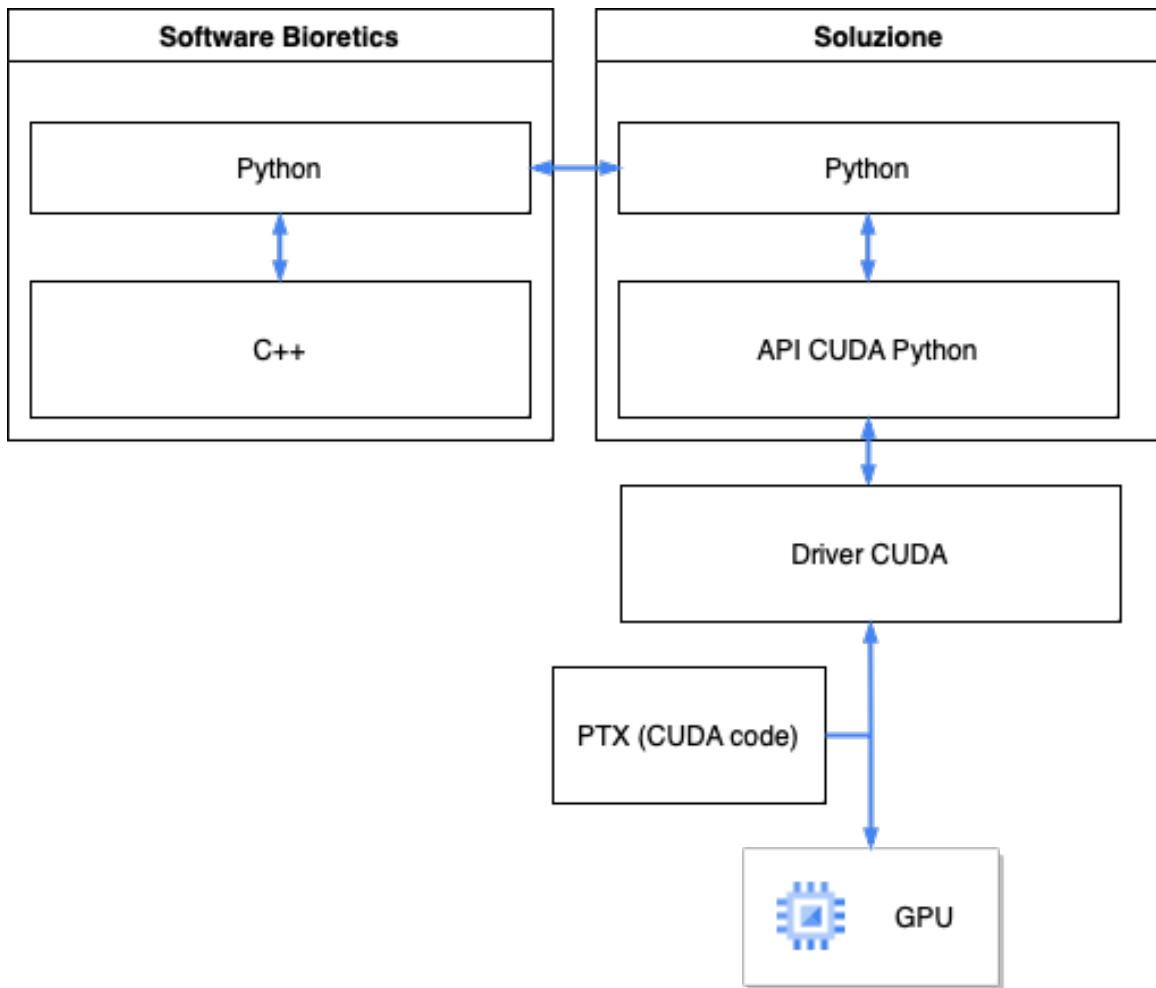


Figura 3.2: Schema utilizzo API CUDA-Python per utilizzo GPU da Python.

La strategia ideata per il problema reale è di creare una funzione Python a cui poter passare il tensore Numpy in cui è memorizzata l'immagine su cui applicare Marching Squares e che restituisca i contorni individuati dall'algoritmo in una struttura dati compatibile nativamente con Python oppure Numpy.

La funzione Python potrà essere importata nel codice dell'azienda ed essere direttamente richiamabile: si occuperà quindi di interfacciarsi con le applicazioni esterne ad alto livello, senza esporre l'implementazione a basso livello del codice CUDA e tutte le procedure necessarie al suo utilizzo. A causa però di alcuni parametri molto specifici relativi alla GPU utilizzata che vanno specificati per una corretta esecuzione ed utilizzo dell'hardware, il codice non potrà rimanere invariato per macchine con schede video aventi caratteristiche particolarmente differenti. I dati dell'immagine passata alla funzione andranno inevitabilmente trasferiti sulla memoria della GPU per poter essere elaborati dai kernel. In seguito andrà portato dalla memoria della scheda video a quella dell'host il risultato ottenuto con l'esecuzione parallela.

### 3.2.1 Strutture dati risultato

La struttura dati utilizzata per il risultato dalla versione seriale di skimage è una lista. Una delle principali sfide per una versione parallela è di riuscire a creare, mantenere e utilizzare strutture dati native C come alternativa. Una lista è una struttura dati estremamente flessibile che può essere creata senza dover specificare a priori la quantità di memoria che dovrà essere utilizzata in seguito, a cui possono essere aggiunti elementi in modo incrementale. Oltre alla lista viene anche utilizzata la struttura dati tupla che può essere simulata in C con `struct`. Ma dato che nella versione di skimage viene utilizzata una tupla contenente due tuple non è possibile creare un array contenente questa struttura dati avendo la certezza che la memoria utilizzata sia contigua. Gli elementi di una tupla vengono disposti in modo contiguo in memoria solo se la loro dimensione è multipla di quella utilizzata dal compilatore altrimenti viene inserito del padding tra un elemento e l'altro. In questo caso il tipo utilizzato è il `double` che occupa 8 byte e potrebbe quindi essere disposto in sequenza in memoria senza l'aggiunta di padding. Per quanto riguarda una tupla contenente due tuple però non è possibile stabilire con certezza come verranno disposti in memoria: non è quindi una soluzione ottimizzata utilizzare un array contenente queste tuple.

Una struttura dati supportata nativamente da C che possa sostituire una lista di tuple contenenti tuple di double sono un insieme di 4 array. Non è la sola a poter esser utilizzata ma è quella che ottimizza maggiormente l'utilizzo dello spazio dei propri elementi e dello spazio totale. Ovviamente mantenere 4 array disconnessi fisicamente ma uniti solo a livello logico è un metodo più delicato e richiede un'attenzione maggiore per mantenerli consistenti. L'idea è di memorizzare i 4 double utilizzando un unico indice per accedere alla stessa rispettiva posizione nei 4 array C.

La struttura dati mantenuta su Python da skimage è rappresentata nella figura 3.3 e quella che verrà utilizzata per la versione parallela nella figura 3.4.

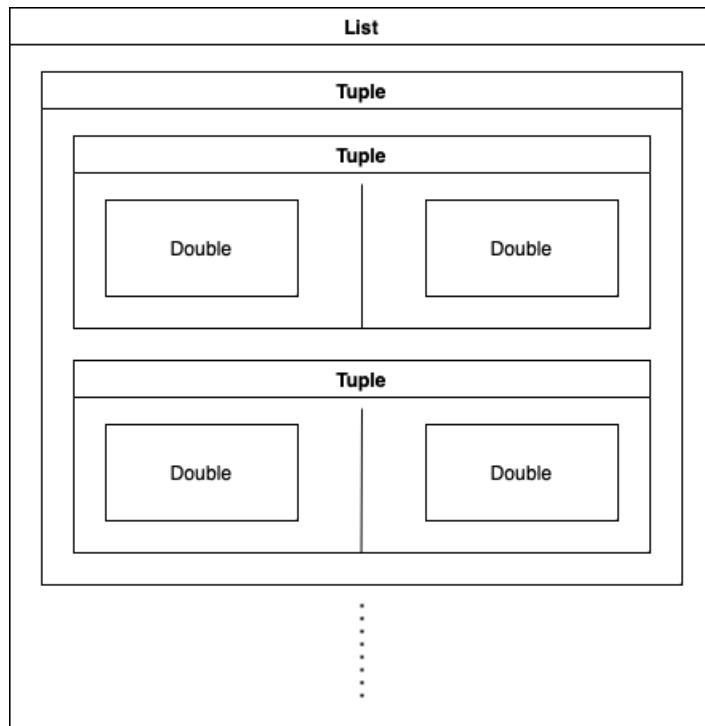


Figura 3.3: Insieme di strutture dati utilizzate per la versione Python di skimage.

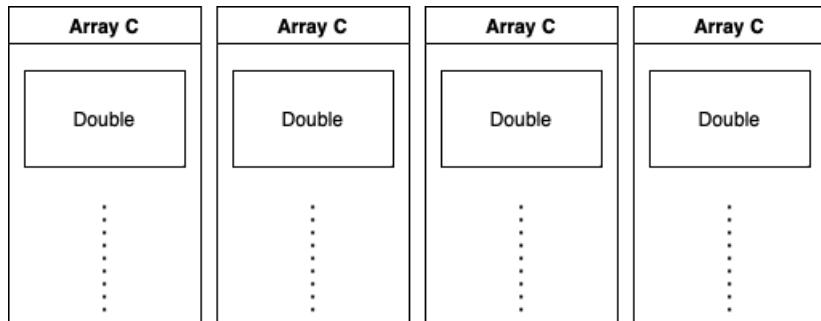


Figura 3.4: Insieme di strutture dati utilizzate per la versione parallela.

Gli array C di tipo double consentono di avere la certezza che i vari elementi dello stesso array siano memorizzati in modo contiguo in memoria. Questa caratteristica è di fondamentale importanza per le operazioni in memoria a cui saranno sottoposti come scritture, letture, trasferimenti da e su GPU; che privilegiano accessi coalizzati ad aree contigue per sfruttare interamente i bus invece che solo parzialmente ricorrendo in una penalizzazione sui tempi.

## 3.3 Kernel Cuda

La fase di progettazione dei kernel Cuda [3] è quella che ha richiesto il maggior studio e pianificazione in quanto la versione seriale del codice skimage sfrutta una struttura dati per l'output che non è utilizzabile in una versione parallela.

Dopo una veloce analisi del codice seriale è chiaro che un singolo kernel non è sufficiente ne sono necessari diversi da lanciare in sequenza per riuscire a replicare il funzionamento del codice seriale in parallelo.

### 3.3.1 Kernel required\_memory

Dato che per utilizzare gli array è necessario definire la dimensione che occuperanno in memoria al momento della loro creazione è necessario stabilire quante posizioni degli array di output verranno utilizzate per una certa immagine.

Il primo kernel quindi si occuperà di calcolare in modalità parallela il numero di segmenti che dovranno essere memorizzati con l'algoritmo Marching Squares per ogni singolo quadrato che andrà esaminato. Per ogni cella verranno stabilite quante posizioni dell'array saranno occupate dai segmenti del contorno che conterrà.

---

#### Codice CUDA kernel required\_memory

---

```
1 size_t r0 = blockIdx.y * blockDim.y + threadIdx.y;
2 size_t c0 = blockIdx.x * blockDim.x + threadIdx.x;
3     /*
4     ...
5     */
6 if (square_case == 0 || square_case == 15){
7     // 0
8     result_required_memory[ r0 * width + c0 ] = 0;
9 }
10 else if (square_case == 6 || square_case == 9){
11     // 2
12     result_required_memory[ r0 * width + c0 ] = 2;
13 }
14 else {
15     // 1
16     result_required_memory[ r0 * width + c0 ] = 1;
17 }
```

---

L'output di questo primo kernel sarà quindi un vettore di dimensione pari al totale delle celle analizzabili dell'immagine e conterrà per ognuno di essi un valore da 0 a 2 che indica quanti segmenti dovrà memorizzare come mostrato nella figura 3.5. I valori per ogni cella sono memorizzati nell'array risultato con una logica posizionale che associa dal primo elemento all'ultimo i valori delle celle dell'immagine scorrendole da sinistra a destra e dall'alto verso il basso.

**Array C**

1	0	0	1	2	1	0	0	0	1
---	---	---	---	---	---	---	---	---	---

Figura 3.5: Array C risultato dell'esecuzione del kernel `required_memory`.

Non abbiamo ancora a disposizione il numero di posizioni totali necessarie per memorizzare tutti i segmenti. Il vettore va immaginato di dimensioni elevate tali per cui una somma dei suoi elementi in versione seriale sia estremamente lenta in relazione alla versione parallela. Sono quindi necessarie altre operazioni parallele per poter ottenere la somma di tutti gli elementi dell'array generato da questo primo kernel.

### 3.3.2 Kernel reduce

Per calcolare la somma di tutti gli elementi dell'array di output del kernel required\_memory è necessario effettuare un'operazione di riduzione che è possibile parallelizzare.

La sum-reduce viene effettuata utilizzando ad ogni passaggio un numero di thread pari alla metà del numero di elementi rimasti. Ad ogni iterazione ogni thread somma al valore presente nella posizione dell'array rispettiva al proprio indice `threadIdx.x` il valore nella metà superiore corrispondente alla stessa posizione ovvero  $(blockDim.x / 2) + threadIdx.x$ .

In questo modo il risultato può essere ottenuto in  $O(\log_2 n)$  passi paralleli come mostrato nell'immagine 3.6.

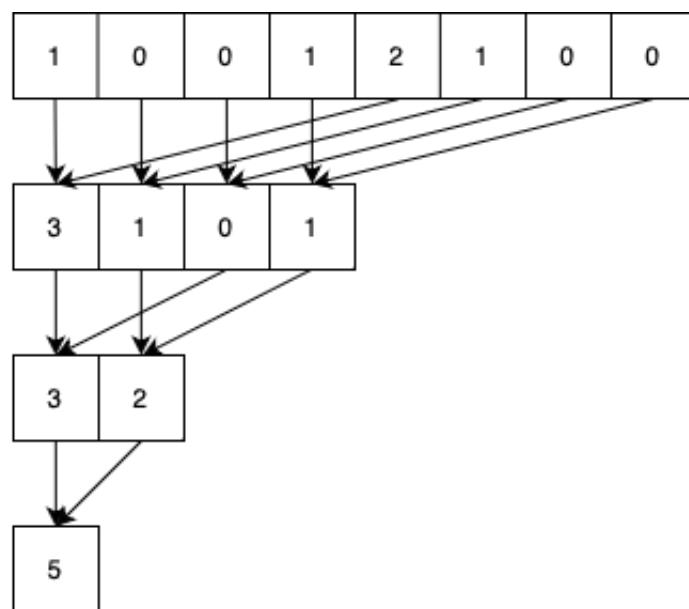


Figura 3.6: Schema operazione di sum-reduce.

Ad ogni passo si dimezzano come il numero di elementi restanti anche il numero di thread utilizzati; questo significa che ci saranno sempre più thread che resteranno inevitabilmente senza alcun lavoro da svolgere nelle iterazioni successive.

Un'accortezza è lanciare il kernel con un numero di thread che sia la metà della lunghezza dell'array in modo che per almeno il primo passo tutti quanti i thread siano utilizzati.

Tra un passo e l'altro è necessario richiamare la funzione `__syncthreads()` che attende tutti i thread in esecuzione, per evitare che i diversi thread lavorino a passi differenti della riduzione rendendo inconsistenti le letture sull'array.

---

## Codice CUDA kernel sum-reduce

---

```
1 const size_t lindex = threadIdx.x;
2 const size_t bindex = blockIdx.x;
3 const size_t gindex = blockIdx.x * blockDim.x + threadIdx.x;
4 size_t bsize = blockDim.x / 2;
5 temp[lindex] = required_memory[gindex];
6 __syncthreads();
7
8 while( bsize > 0 ){
9     if( lindex < bsize && (lindex+bsize)<n ){
10         temp[lindex] += temp[lindex+bsize];
11     }
12     bsize = bsize / 2;
13     __syncthreads();
14 }
15 if(0==lindex){
16     result_reduce[bindex] = temp[0];
17 }
```

---

In seguito alla lettura iniziale dalla memoria principale della GPU, tutte le operazioni vengono effettuate successivamente sono sulla memoria condivisa della scheda video che richiede tempi molto più rapidi per le operazioni di lettura e scrittura.

Lo scalare ottenuto con l'ultimo passo viene copiato dal thread 0 nella posizione relativa al proprio blocco sull'array risultato. Ogni blocco di thread ottiene quindi la somma dei valori corrispondenti alle sue posizioni; una volta riportati questi risultati parziali sull'host è necessario effettuare un'ulteriore operazione di somma su di essi per ottenere il risultato finale. In questo caso però il numero di thread necessari per elaborare tutto l'array non è sufficientemente grande da garantire una convenienza in termini temporali per il lancio di un kernel parallelo su GPU. Conviene effettuare la somma degli elementi sull'host con una funzione ottimizzata come `.sum()` fornita da Numpy.

Il risultato finale che unisce i risultati parziali degli altri blocchi è ottenuto con il seguente codice Python dall'host.

---

## Codice host in Python per somma risultati parziali del kernel reduce

---

```
1 np_result_reduce = np.array(result_reduce)
2 N_RES = np_result_reduce.sum()
```

---

### 3.3.3 Kernel exclusive scan (prescan)

Una volta ricavato quanto sia il totale di segmenti che andranno memorizzati solo un altro componente per poter lanciare il kernel che eseguirà Marching Squares.

Ora che possiamo allocare tutto lo spazio necessario e che tutti i thread che eseguiranno MS avranno uno spazio sufficiente per scrivere i segmenti trovati, è necessario definire in quali posizioni della memoria a loro riservata ognuno dovrà scrivere.

Esiste una funzione da applicare a un array che calcola esattamente il risultato di cui abbiamo bisogno, ovvero un array che contiene le posizioni in cui il thread corrispondente alla posizione in cui è salvata deve scrivere i suoi risultati. Questa funzione è la exclusive scan (chiamata anche prescan), che calcola tutti i prefissi di un array in base ad una operazione binaria data che nel nostro caso è quella di somma.

La principale caratteristica dell'exclusive scan è che nella prima posizione dell'array risultato viene messo il valore dell'elemento neutrale dell'operazione utilizzata (lo zero per la somma). Tutti i valori del risultato sono quindi spostati di una posizione verso destra e l'ultimo valore dell'array in input non viene considerato. Un esempio di array risultato prodotto dalla exclusive scan è riportato nella figura 3.7.

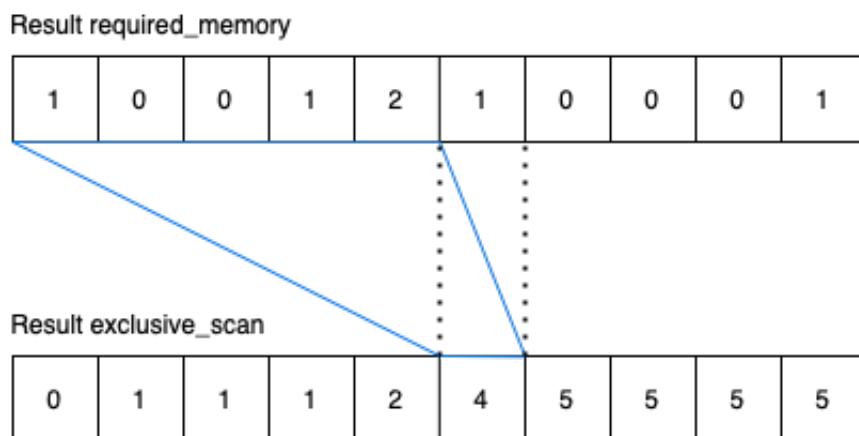


Figura 3.7: Vettori di input e output della funzione di exclusive scan.

Il risultato di una exclusive scan per le sue proprietà è già pronto a essere utilizzato per lo scopo desiderato: in ogni cella infatti è contenuta la posizione in cui andrà scritto il rispettivo valore. Rispetta anche gli indici degli array siccome la prima posizione in cui scrivere indicata è lo 0. Il valore della posizione non considera quanti segmenti andranno scritti dal corrispondente thread, ma solo quelli precedenti poiché è necessario saper solo da dove iniziare a scrivere e non dove si finirà.

Sono presenti valori ripetuti nel vettore in output dato che l'indice della prima posizione disponibile in cui scrivere non deve variare se i thread successivi non hanno alcun segmento da memorizzare. I thread che invece troveranno 1 o 2 segmenti dovranno scriverli consecutivamente, partendo dalla posizione specificata nel risultato dell'exclusive scan.

Per effettuare l'operazione di exclusive scan su un array di grandi dimensioni un singolo kernel non è sufficiente. Il codice consigliato da Nvidia per implementare la Exclusive Scan [1] funziona solamente se l'array su cui si deve applicare la funzione può essere contenuto all'interno della memoria condivisa da un singolo blocco della GPU; inoltre può utilizzare solamente la metà del numero massimo di thread per un blocco.

Per comporre il vettore finale in modalità parallela è stato necessario implementare un totale di 3 kernel differenti:

- prescan (exclusive scan)
- prescan\_small (exclusive scan small)
- add

Questa particolare versione permette l'applicazione della exclusive scan ad array di lunghezza arbitraria anche maggiore alla memoria condivisa di un singolo blocco.

In pratica il kernel prescan esegue una exclusive scan su tutto l'array ma a comparti di 64 elementi; ogni 64 posizioni dell'array principale il conteggio riparte da zero.

Nella figura 3.8 è rappresentato un esempio del risultato prodotto dal kernel `prescan` con il numero di elementi per blocco ridotto da 64 a 20 per semplicità.

res_prescan										
0	0	1	1	1	1	1	2	2	2	
2	2	3	3	4	4	4	4	4	4	4
0	0	0	1	1	1	1	1	1	1	
1	1	1	2	2	2	3	3	3	3	3
0	1	1	1	2	2	3	3	3	3	

Figura 3.8: Esempio di risultato del kernel prescan con blocchi da 20 elementi invece di 64.

Sempre all'interno del kernel `prescan` si salva l'ultimo valore di ogni blocco (colorato in blu nella figura 3.8) cioè la somma di tutti i valori che contiene (il 64° elemento) in un altro array chiamato `sums` con dimensione pari al numero di blocchi utilizzato. Se si facesse riferimento all'esempio della figura 3.8 il vettore `sums` conterebbe nelle prime due posizioni i valori 4 e 3. Si prosegue poi con il kernel `prescan_small` che lavora con 1 solo blocco di thread ed effettua una exclusive scan sull'array `sums`. Infine si utilizza un ulteriore kernel denominato `add` che si occupa di sommare a `res_prescan` citato sopra il rispettivo valore contenuto nel risultato del kernel `prescan_small` (che sostanzialmente è la somma di tutti i valori dei compartimenti da 64 precedenti) per compartimenti da 64.

Nella seguente figura 3.9 viene rappresentato in modo semplificato in che modo i tre kernel collaborano per ottenere il risultato dell'exclusive scan.

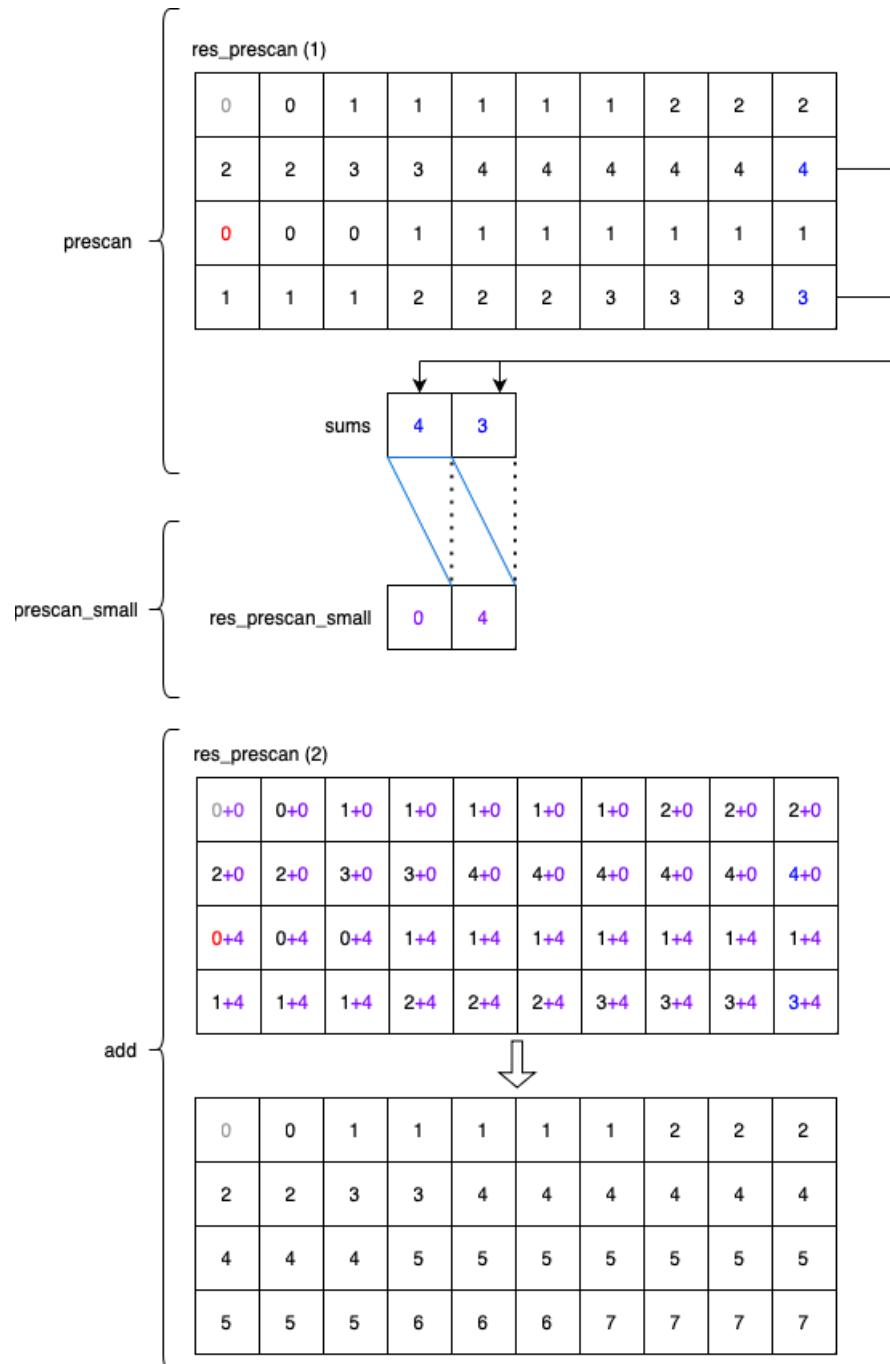


Figura 3.9: Esempio rappresentativo delle trasformazioni applicate dai tre kernel per la funzione exclusive scan.

### 3.3.4 Kernel marching\_squares

Con a disposizione il risultato della Exclusive scan si può procedere con il kernel `marching_squares`. L'array `res_prescan` contiene le posizioni da cui i thread corrispondenti ad ogni suo elemento dovranno iniziare a scrivere i segmenti che troveranno nell'esecuzione del kernel `marching_squares`. Questo kernel è in sostanza il corrispondente CUDA del codice seriale Cython sviluppato da skimage. La principale differenza è la scrittura dei segmenti del risultato, dove si è passati dal semplice `append(tuple(tuple(),tuple()))` su una lista, alla scrittura su 4 array nelle posizioni individuate per ogni thread dai kernel precedenti.

Ogni thread che esegue questo kernel si occupa di una singola cella dell'immagine in input e svolge questi passaggi:

1. Tramite la logica di Marching Squares individua a quale dei 16 possibili tipi appartiene la cella
2. Accede all'array restituito dalla exclusive scan nella posizione corrispondente al proprio blocco e id per ottenere la posizione in cui dovrà scrivere il risultato
3. Se il tipo di cella è diverso da 0 o 15 allora scrive nell'array risultato le coordinate degli estremi dei segmenti trovati partendo dalla posizione recuperata in precedenza

Tutti i kernel precedenti consentono a questo kernel finale di poter scrivere sull'array risultato (della dimensione esattamente necessaria) in maniera parallela.

Di seguito è riportato il codice completo del kernel `marching_squares` esclusa la funzione `get_fraction` che è l'equivalente CUDA della versione Cython riportata in precedenza.

Come si può vede dalle righe 4 e 5 ad ogni thread viene assegnata la corrispondente cella, il quale spigolo in alto a sinistra coincide come posizione nella matrice alla combinazione di blocco e id del thread sugli assi  $x$  e  $y$ .

---

## Codice kernel marching\_squares

---

```
1 void marching_squares(double *image, double *result_1x, double *result_1y,
2   double *result_2x, double *result_2y, double level, int n, int width,
3   int height, int *positions)
4 {
5   int square_case;
6   int r0 = blockIdx.y * blockDim.y + threadIdx.y;
7   int c0 = blockIdx.x * blockDim.x + threadIdx.x;
8   int r1 = r0 + 1;
9   int c1 = c0 + 1;
10  struct tuple {
11    double x;
12    double y;
13  } top, bottom, left, right;
14
15  if( r0 < height-1 && c0 < width-1 ){
16    double ul = image[ r0 * width + c0 ];
17    double ur = image[ r0 * width + c1 ];
18    double ll = image[ r1 * width + c0 ];
19    double lr = image[ r1 * width + c1 ];
20
21    square_case = 0;
22    if (ul > level) square_case += 1;
23    if (ur > level) square_case += 2;
24    if (ll > level) square_case += 4;
25    if (lr > level) square_case += 8;
26
27    // determinate the position of the result array where to write the
28    // number of values every thread needs
29    int g-pos = positions[r0 * width + c0];
30
31    if (square_case != 0 && square_case != 15){
32      // case 0 and 15 have no values to write
33      top.x = r0;
34      top.y = c0 + get_fraction(ul,ur,level);
35      bottom.x = r1;
36      bottom.y = c0 + get_fraction(ll,lr,level);
37      left.x = r0 + get_fraction(ul,ll,level);
38      left.y = c0;
39      right.x = r0 + get_fraction(ur,lr,level);
40      right.y = c1;
41
42      if (square_case == 1){
43        result_1x[ g-pos ] = top.x;
44        result_1y[ g-pos ] = top.y;
```

```

43         result_2x[ g-pos ] = left.x;
44         result_2y[ g-pos ] = left.y;
45     }
46     else if (square_case == 2){
47         result_1x[ g-pos ] = right.x;
48         result_1y[ g-pos ] = right.y;
49         result_2x[ g-pos ] = top.x;
50         result_2y[ g-pos ] = top.y;
51     }
52     else if (square_case == 3){
53         result_1x[ g-pos ] = right.x;
54         result_1y[ g-pos ] = right.y;
55         result_2x[ g-pos ] = left.x;
56         result_2y[ g-pos ] = left.y;
57     }
58     else if (square_case == 4){
59         result_1x[ g-pos ] = left.x;
60         result_1y[ g-pos ] = left.y;
61         result_2x[ g-pos ] = bottom.x;
62         result_2y[ g-pos ] = bottom.y;
63     }
64     else if (square_case == 5){
65         result_1x[ g-pos ] = top.x;
66         result_1y[ g-pos ] = top.y;
67         result_2x[ g-pos ] = bottom.x;
68         result_2y[ g-pos ] = bottom.y;
69     }
70     else if (square_case == 6){
71         // 2 couple of points to write
72         result_1x[ g-pos ] = left.x;
73         result_1y[ g-pos ] = left.y;
74         result_2x[ g-pos ] = top.x;
75         result_2y[ g-pos ] = top.y;
76
77         result_1x[ g-pos + 1 ] = right.x;
78         result_1y[ g-pos + 1 ] = right.y;
79         result_2x[ g-pos + 1 ] = bottom.x;
80         result_2y[ g-pos + 1 ] = bottom.y;
81     }
82     else if (square_case == 7){
83         result_1x[ g-pos ] = right.x;
84         result_1y[ g-pos ] = right.y;
85         result_2x[ g-pos ] = bottom.x;
86         result_2y[ g-pos ] = bottom.y;
87     }
88     else if (square_case == 8){
89         result_1x[ g-pos ] = bottom.x;
90         result_1y[ g-pos ] = bottom.y;
91         result_2x[ g-pos ] = right.x;

```

```

92         result_2y[ g-pos ] = right.y;
93     }
94     else if (square_case == 9){
95         // 2 couple of points to write
96         result_1x[ g-pos ] = top.x;
97         result_1y[ g-pos ] = top.y;
98         result_2x[ g-pos ] = right.x;
99         result_2y[ g-pos ] = right.y;
100
101        result_1x[ g-pos + 1 ] = bottom.x;
102        result_1y[ g-pos + 1 ] = bottom.y;
103        result_2x[ g-pos + 1 ] = left.x;
104        result_2y[ g-pos + 1 ] = left.y;
105    }
106    else if (square_case == 10){
107        result_1x[ g-pos ] = bottom.x;
108        result_1y[ g-pos ] = bottom.y;
109        result_2x[ g-pos ] = top.x;
110        result_2y[ g-pos ] = top.y;
111    }
112    else if (square_case == 11){
113        result_1x[ g-pos ] = bottom.x;
114        result_1y[ g-pos ] = bottom.y;
115        result_2x[ g-pos ] = left.x;
116        result_2y[ g-pos ] = left.y;
117    }
118    else if (square_case == 12){
119        result_1x[ g-pos ] = left.x;
120        result_1y[ g-pos ] = left.y;
121        result_2x[ g-pos ] = right.x;
122        result_2y[ g-pos ] = right.y;
123    }
124    else if (square_case == 13){
125        result_1x[ g-pos ] = top.x;
126        result_1y[ g-pos ] = top.y;
127        result_2x[ g-pos ] = right.x;
128        result_2y[ g-pos ] = right.y;
129    }
130    else if (square_case == 14){
131        result_1x[ g-pos ] = left.x;
132        result_1y[ g-pos ] = left.y;
133        result_2x[ g-pos ] = top.x;
134        result_2y[ g-pos ] = top.y;
135    }
136}
137}
138}

```

---

## 3.4 Codice Python con API CUDA Python

Per poter lanciare i kernel CUDA da Python è necessario utilizzare le API CUDA Python come descritto in precedenza.

Questo componente serve da interfaccia per le chiamate Python esterne e si occupa di tutto il necessario per poter lanciare i kernel sopra descritti ottenendo il risultato richiesto.

Questo modulo è costituito principalmente da codice molto simile a quello descritto nella sezione 3.1.2. Tutte le logiche di preparazione e lancio di un kernel sono però moltiplicate per 6; ovvero il numero di kernel totali che sono necessari come descritto in precedenza.

Per motivi di praticità, ci limitiamo a riportare e analizzare solo le parti più significative.

Tra un kernel e l'altro la funzione `cuda.cuStreamSynchronize(stream)` indicata da Nvidia non si è rivelata sufficiente per una corretta esecuzione: si presentava infatti un `illegal access error` che terminava l'esecuzione. Un errore di questo tipo solitamente è causato da un errore all'interno del codice ma in questo caso si è scoperto essere dovuto a un problema delle API Cuda Python, che faticano a gestire chiamate consecutive senza pause generando ritardi nella risoluzione delle chiamate ai driver Nvidia. Per risolvere il problema è stato necessario effettuare numerose prove che hanno condotto all'aggiunta di una ulteriore funzione in accoppiata a `cuda.cuStreamSynchronize(stream)` che permette di risolvere il problema. Come si può vedere nel seguente estratto di codice è stata aggiunta la sincronizzazione anche del `context` oltre che a quella dello `stream`.

---

### Codice migliorato per sincronizzazione con API CUDA Python

---

```
1 # For Illegal memory access error
2 err , = cuda.cuCtxSynchronize()
3 ASSERTDRV(err)
4 err , = cuda.cuStreamSynchronize(stream)
5 ASSERTDRV(err)
```

---

Per quanto riguarda invece il lancio del kernel finale `marching_squares` va fatta un'osservazione sulla corrispondenza tra le celle dell'immagine da esaminare con MS e i thread che se ne occupano. Come analizzato in precedenza nella sezione del kernel `marching_squares` ad ogni thread viene assegnata la cella dell'immagine il cui spigolo in alto a sinistra corrisponde come posizione nella matrice. Di conseguenza il numero di pixel sarà maggiore del numero di thread necessari, per esattezza il numero di thread sarà  $numero\_pixel - altezza\_immagine - larghezza\_immagine + 1$ . Il calcolo è facilmente intuibile dalla figura 3.10 che riporta l'associazione tra un thread e una singola cella composta da 4 pixel sulla sinistra e mostra quanti thread sono necessari (celle in azzurro) per coprire una certa matrice di pixel sulla destra.

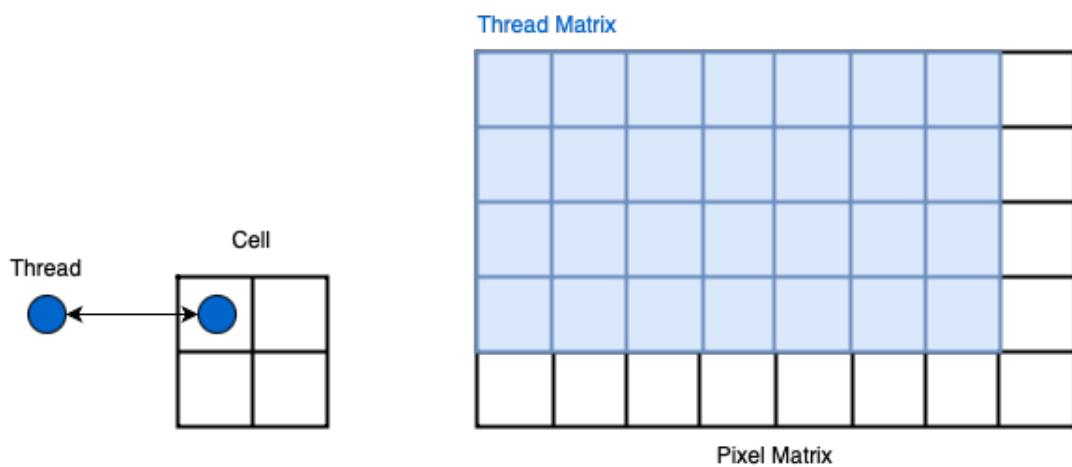


Figura 3.10: Corrispondenza tra thread e cella composta da 4 pixel.

Gli array necessari ai kernel per input o output vengono creati nella fase iniziale prima del lancio dei kernel. Ci sono però degli array che per motivi di logica della soluzione parallela non posso seguire questa sequenza. I quattro array del risultato infatti possono essere creati solo dopo aver lanciato i primi kernel poiché è necessario sapere il risultato della sum-reduce per avere la dimensione corretta con cui devono essere allocati. Prima dell'ultimo kernel `marching_squares` vengono quindi creati i quattro array utilizzando il risultato ottenuto dall'elaborazione del risultato parziale della sum-reduce come riportato nel seguente estratto di codice.

---

### Estratto di codice con allocazione quattro array per output del kernel `marching_squares` calcolando risultato finale di sum-reduce

---

```

1 np_result_reduce = np.array(result_reduce)
2 N_RES = np_result_reduce.sum()
3
4 bufferSize_resultsMS = N_RES * lev_np.itemsize
5
6 result_1x = np.zeros(N_RES).astype(dtype=np.float64)
7 result_1y = np.zeros(N_RES).astype(dtype=np.float64)
8 result_2x = np.zeros(N_RES).astype(dtype=np.float64)
9 result_2y = np.zeros(N_RES).astype(dtype=np.float64)
10
11 err, dResult1Xclass = cuda.cuMemAlloc(bufferSize_resultsMS)
12 ASSERTDRV(err)
13 err, dResult1Yclass = cuda.cuMemAlloc(bufferSize_resultsMS)
14 ASSERTDRV(err)
15 err, dResult2Xclass = cuda.cuMemAlloc(bufferSize_resultsMS)
16 ASSERTDRV(err)
17 err, dResult2Yclass = cuda.cuMemAlloc(bufferSize_resultsMS)
18 ASSERTDRV(err)
19
20 dResult_1x = np.array([int(dResult1Xclass)], dtype=np.uint64)
21 dResult_1y = np.array([int(dResult1Yclass)], dtype=np.uint64)
22 dResult_2x = np.array([int(dResult2Xclass)], dtype=np.uint64)
23 dResult_2y = np.array([int(dResult2Yclass)], dtype=np.uint64)
24
25 args_6 = [dImage, dResult_1x, dResult_1y, dResult_2x, dResult_2y, lev_np,
26           n, width, height, dResult_exc_scan]
26 args_6 = np.array([arg.ctypes.data for arg in args_6], dtype=np.uint64)
```

---

Il kernel `reduce` risulta fondamentale dato che non è possibile sapere a priori quanti segmenti è necessario memorizzare senza di esso. Neanche il numero di celle totali dell'immagine sarebbe un limite superiore: potrebbe succedere infatti che se più della metà delle celle appartenessero ai tipi 6 o 9 di quelle predefinite da MS allora in numero totale di segmenti sarebbe superiore. Inoltre creare array di dimensioni maggiori di quelle necessarie indurrebbe un overhead causato dai tempi maggiori di trasferimento degli array tra le memorie.

# Capitolo 4

## Risultati ottenuti

Il codice implementato è in grado di applicare l'algoritmo Marching Squares in parallelo su GPU impiegando un tempo inferiore rispetto al metodo `find_contours` della libreria skimage. I test sono stati effettuati su due diversi tipi di immagini in input, una foto reale scattata da un macchina per la selezione della frutta e una generata artificialmente con delle funzioni matematiche. Tutti i test seguenti sono stati svolti su una macchina con una Nvidia 1050 Ti.

### 4.1 Immagine reale

L'immagine in questione è stata fornita da Bioretics e contiene una quantità di bordi che rappresenta indicativamente la media individuabile in un set di scatti. Oltre ad individuare i contorni dei difetti del frutto vengono esaminate anche tutte le altre aree dell'immagine comprese alcune zone dei rulli in cui la CNN ha attribuito una classe (anche la classe "rullo" viene assegnata). L'immagine utilizzata per questi test è riportata nella figura 4.1.



Figura 4.1: Fotografia reale utilizzata per i test.

Questo preciso scatto ritrae un'albicocca che ruota sui nastri su cui la CNN ha individuato un difetto sulla buccia nella parte alta del frutto e il picciolo. Sono state trovate anche molte zone che fanno parte della struttura o dei rulli stessi. La foto viene scattata inquadrando un'area così abbondante dei rulli poiché sarebbe troppo complesso individuare la posizione esatta in cui si troverà un certo frutto che scorre e rotola a velocità elevata davanti alla camera.

Prima di essere passata al metodo che applica MS l'immagine viene ridimensionata notevolmente e convertita in scala di grigi (grayscale). L'immagine in input al test effettuato sul codice è riportato in figura 4.1. L'output è stato riportato in formato grafico nell'immagine 4.3 in cui si possono notare dei bordi colorati che sono stati ricostruiti attorno alle aree con valore maggiore della soglia (impostata in questo caso a 0.5).



Figura 4.2: Immagine 4.1 ridimensionata e convertita in grayscale.

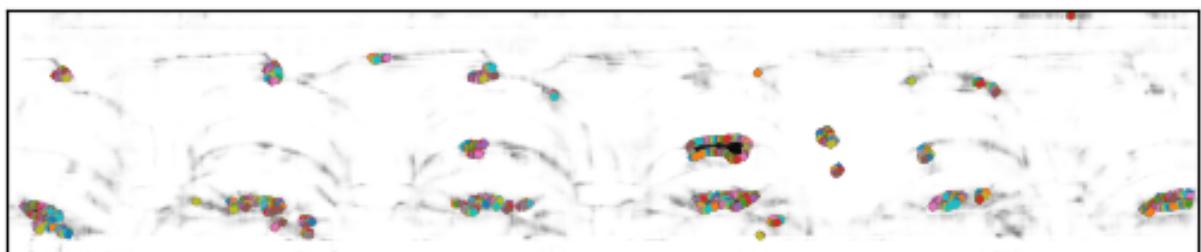


Figura 4.3: Immagine 4.2 con disegnati i contorni individuati.

Sono riportati quindi nelle immagine 4.5 i contorni relativi al difetto e del picciolo del frutto dell'immagine 4.4.

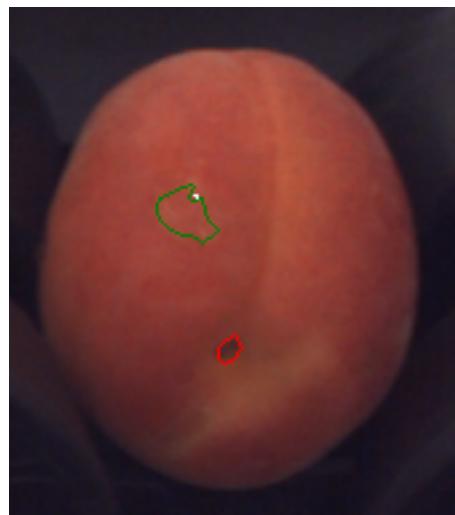


Figura 4.4: Foto del frutto (albicocca) estratto dall'immagine 3.11.

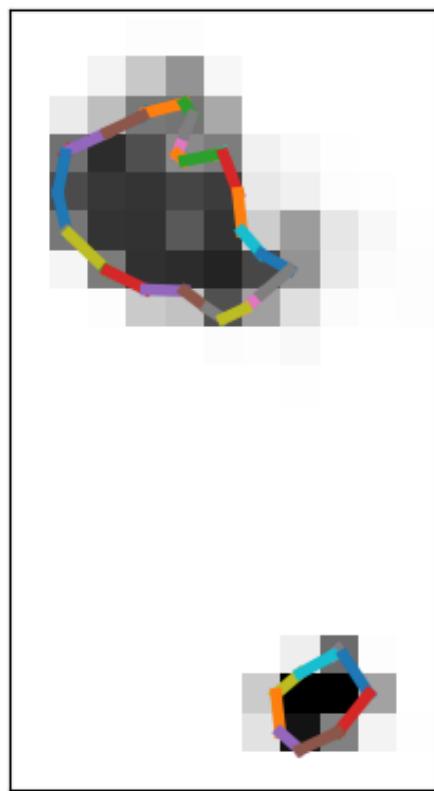


Figura 4.5: Ingrandimento di una sezione dell'immagine 3.13.

In termini di prestazioni sono stati registrati tempi di esecuzione minori di quelli impiegati dalla funzione `find_contours`. Si è quindi ottenuto uno speedup raggiungendo l'obbiettivo del progetto, va però fatta una importante osservazione sul metodo di misura di questo speedup. La funzione `find_contours` è costituita da due componenti principali come descritto in precedenza, la seconda parte `_assemble_contours` che si occupa della congiunzione dei segmenti di contorno adiacenti, dovrebbe essere idealmente esclusa dalla misurazione dei tempi di esecuzione. Non è possibile farlo però in maniera esatta poiché la chiamata a `_assemble_contours` viene fatta all'interno del metodo `find_contours` di cui è possibile misurare solo i tempi di esecuzione totali. Dato che il codice della libreria skimage è pubblico e reperibile è stato possibile fare un test indicativo dei tempi di esecuzione di `_assemble_contours` testandoli sul singolo metodo con lo stesso input che avrebbe ricevuto se richiamato all'interno di `find_contours`. I tempi di esecuzione misurati per le due diverse versioni e quello calcolato come stima del metodo `_assemble_contours` sono riportati nella seguente tabella 4.1.

Tempi di esecuzione medi	
Metodo <code>find_contours</code> (skimage)	0.001192593 s
Metodo <code>assemble_contours</code> (skimage)	0.000629877 s
Metodo con API CUDA-Python	0.000283867 s

Tabella 4.1: Confronto tempi (per immagine reale) di esecuzione versione skimage ovvero il tempo totale impiegato dal metodo `find_contours`, quello impiegato dal singolo metodo `assemble_contours` e quello relativo alla versione parallela su GPU che sfrutta le API CUDA-Python.

I tempi sono stati riportati graficamente nella figura 4.6. Nella barra che rappresenta la versione skimage ovvero tutto il metodo `find_contours` è stata colorata in azzurro più chiaro la porzione che si è stimato impieghi il metodo `_assemble_contours`. Come si può notare, la versione parallela sviluppata con le API CUDA-Python applica l'algoritmo Marching Squares impiegando un tempo minore rispetto alla versione skimage con un ampio margine.

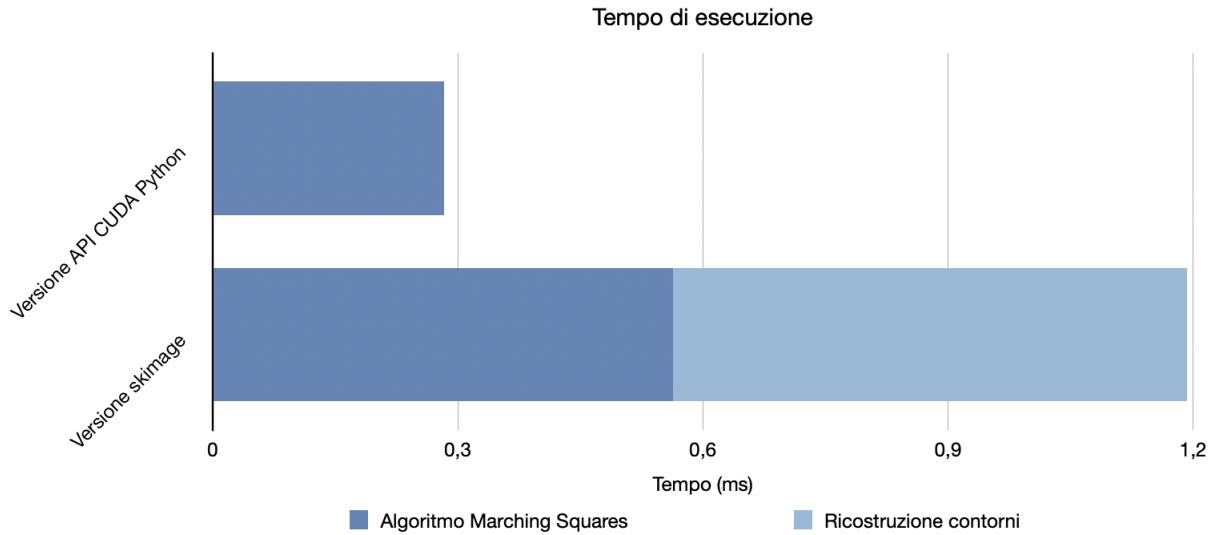


Figura 4.6: Tempi di esecuzione delle due metodi analizzati misurati sull’immagine reale.

Si possono fare anche osservazioni sul codice sviluppato da skimage, che presenta delle scelte progettuali estremamente distanti tra loro nei due metodi che costituiscono `find_contours`. Il metodo interno `_get_contour_segments` che applica l’algoritmo Marching Squares è stato particolarmente ottimizzato con l’utilizzo del Cython in modo da ridurre i tempi al minimo. Il metodo `_assemble_contours` invece è composto quasi interamente da pesanti operazioni su liste Python che sono estremamente dispendiose in termini di tempo di esecuzione. Come si può vedere dal grafico la stima del tempo impiegato per l’esecuzione di `_assemble_contours` occupa circa la metà del tempo totale medio impiegato da `find_contours`.

Gli speedup della versione parallela calcolati sulla base delle misurazioni precedenti sono stati riportati nella tabella 4.2. Per completezza è stato inserito anche lo speedup misurato sul tempo totale di `find_contours` dato che la misura del tempo impiegato da `_assemble_contours` che è stato sottratto a quello di `find_contours` per il calcolo dello speedup è una stima.

	Speedup stimato
Rispetto <code>find_contours</code>	4.201232
Rispetto <code>find_contours</code> escluso <code>assemble_contours</code>	1.982321

Tabella 4.2: Speedup stimato (per immagine reale) della versione con API CUDA-Python rispetto al metodo `find_contours` considerando il tempo di esecuzione integrale e quella a cui è stata sottratta la stima del tempo impiegato dal suo metodo interno `_assemble_contours`.

Il valore più importante che determina il successo del progetto è quello dello speedup della versione parallela sviluppata rispetto alla porzione di codice skimage sull'esecuzione dell'algoritmo Marching Squares. Seppur sia derivato da una approssimazione del tempo di esecuzione di `_assemble_contours` risulta essere di circa 2 volte. Uno speedup di 2 è sufficientemente grande da poter affermare che uno speedup sia stato raggiunto anche considerando una approssimazione per eccesso dei tempi stimati per `_assemble_contours`.

## 4.2 Immagine artificiale

Le dimensioni dell'immagine generata artificialmente sono le stesse di quella reale ovvero  $511 \times 95$ . Questa immagine però ha un numero maggiore di segmenti necessari per definire i contorni, ne consegue un maggior carico di lavoro per l'algoritmo Marching Squares che deve gestire più casi di quelli che necessitano di un segmento di contorno. L'immagine con cui sono stati effettuati i test è stata creata con l'utilizzo di funzioni matematiche ed è riportato nella figura 4.7. L'output del codice parallelo a cui è stata passato in input questa immagine è osservabile dalla figura 4.8 sottostante.



Figura 4.7: Immagine artificiale generata con funzioni matematiche.

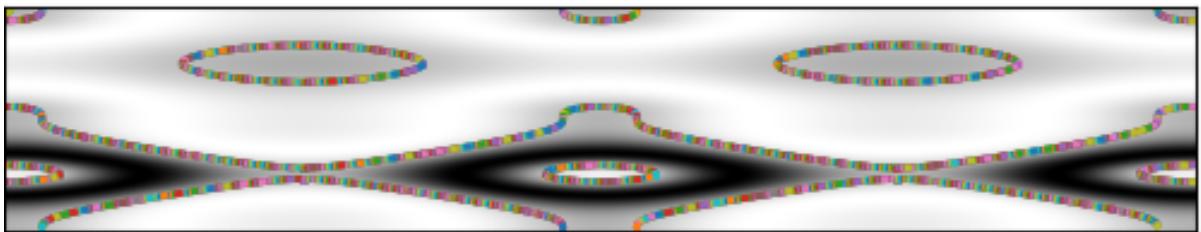


Figura 4.8: Immagine 4.7 con disegnati i contorni individuati.

I tempi di esecuzione sono stati raccolti con lo stesso metodo descritto per l'immagine reale, con la sola differenza che in input alle funzioni è stata passata l'immagine artificiale. La tabella 4.3 riporta i tre tempi misurati.

Tempi di esecuzione medi	
Metodo find_contours (skimage)	0.002102 s
Metodo assemble_contours (skimage)	0.001288 s
Metodo con API CUDA-Python	0.000310 s

Tabella 4.3: Confronto tempi di esecuzione (per immagine artificiale) versione skimage ovvero il tempo totale impiegato dal metodo `find_contours`, quello impiegato dal singolo metodo `assemble_contours` e quello relativo alla versione parallela su GPU che sfrutta le API CUDA-Python.

Anche in questo caso sono stati rappresentati in maniera grafica i tempi di esecuzione delle due versioni consultabili dall'immagine 4.9. In questo caso la porzione della versione skimage occupata da `_assemble_contours` risulta in proporzione ancora maggiore del totale di `find_contours` rispetto a quanto visto con l'immagine reale. Il motivo è dato dal numero maggiore di segmenti individuati per i contorni che mettono in evidenza la scarsa efficienza della funzione `_assemble_contours`. Il tempo di esecuzione impiegato dalla versione parallela di MS con le API CUDA-Python risulta anche in questo caso inferiore di quella di skimage in un rapporto ancora maggiore rispetto a quello riscontrato con l'immagine reale.

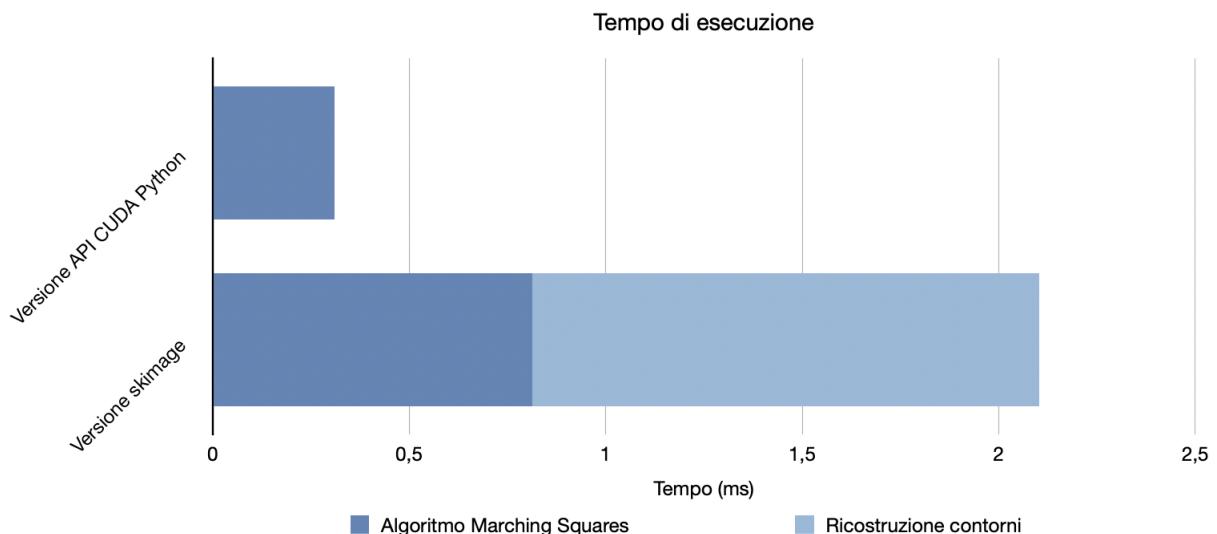


Figura 4.9: Tempi di esecuzione delle due metodi analizzati misurati sull'immagine artificiale.

Per l'immagine artificiale il calcolo dello speedup è stato effettuato nello stesso modo utilizzato in precedenza per l'immagine reale e valgono le stesse osservazioni sulla completezza dei valori riportati nella tabella 4.4.

	Speedup stimato
Rispetto <code>find_contours</code>	6.766283
Rispetto <code>find_contours</code> escluso <code>assemble_contours</code>	2.620361

Tabella 4.4: Speedup stimato (per immagine artificiale) della versione con API CUDA-Python rispetto `find_contours` considerando il tempo di esecuzione integrale e quella a cui è stata sottratta la stima del tempo impiegato dal suo metodo interno `_assemble_contours`.

Lo speedup ottenuto rispetto all'esecuzione di MS interna a `find_contours` risulta maggiore di quello calcolato per l'immagine reale. Con l'immagine artificiale si è ottenuto infatti uno speedup pari a circa 2.6 volte, un risultato che indica un aumento dello speedup in relazione ad un aumento del numero di segmenti come ci si aspettava.

# Capitolo 5

## Conclusioni

L'algoritmo Marching Squares è stato parallelizzato su GPU partendo da codice Python, ottenendo uno speedup di circa 2 volte rispetto alla funzione `find_contours` sulle immagini (reali) acquisite in ambito industriale. L'obbiettivo prefissato per il progetto è stato quindi raggiunto con successo.

La soluzione finale è costituita da un modulo Python che sfrutta le API CUDA Python per interagire con i driver Nvidia e lanciare su GPU i sei kernel CUDA implementati. Sono state fatte anche minuziose ispezioni del codice sviluppato dalla libreria scikit-image (`skimage`) che hanno rivelato livelli di ottimizzazione molto differenti tra i due componenti interni a `find_contours`. I tempi di esecuzione del metodo che hanno dedicato a MS è stato particolarmente ridotto con l'utilizzo del Cython come sostituto del Python per avvicinarsi ai tempi di esecuzione del codice C. Il metodo che congiunge i segmenti di contorno invece è scritto in Python e contiene operazioni su liste che sono estremamente dispendiose in termini di tempo di esecuzione. L'unione di questi due componenti porta `find_contours` a impiegare in certi casi più tempo a unire i segmenti di contorno trovati che a trovarli applicando MS. Se la libreria volesse ridurre ulteriormente il tempo di esecuzione totale dovrebbe indirizzare le proprie risorse sulla fase di ricostruzione dei contorni piuttosto che migliorare ulteriormente la parte di MS.

Per quanto riguarda la soluzione implementata, le API CUDA Python si sono rivelate uno strumento solido nonostante siano state rilasciate di recente. Per apprenderne il funzionamento è stato necessario un tempo ragionevole basandosi sulle conoscenze in merito alla programmazione CUDA e studiando la documentazione fornita da Nvidia.

La composizione dei kernel implementati definisce un pattern di programmazione parallela, che permette di arrivare all'esecuzione di un kernel in cui ogni thread determina quanta memoria necessita per scrivere il proprio risultato solo a runtime. I kernel che sono stati progettati sono in grado di calcolare il totale di memoria necessario e le posizioni da cui ogni thread dovrà iniziare a scrivere i propri risultati: permettono quindi di allocare strutture dati della dimensione necessaria e definiscono anche come dovrà essere riempita dal kernel finale.

Tutti i test e il codice sviluppato sono reperibili al link:

[https://github.com/AlessandroSciarrillo/marching\\_squares\\_parallel\\_GPU](https://github.com/AlessandroSciarrillo/marching_squares_parallel_GPU).

# **Ringraziamenti**

# Bibliografia

- [1] Mark Harris, Shubhabrata Sengupta, John D. Owens. Parallel Prefix Sum (Scan) with CUDA. URL: <https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda>
- [2] Ashwin Srinath: Accelerating Python on GPUs with nvc++ and Cython. URL: <https://developer.nvidia.com/blog/accelerating-python-on-gpus-with-nvc-and-cython/>
- [3] Nvidia. CUDA C++ Programming Guide: <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>
- [4] Nvidia. CUDA Python Documentation: <https://nvidia.github.io/cuda-python/>
- [5] GitHub repositories.  
findContours.py [commit-a91f4f8]: [https://github.com/scikit-image/scikit-image/blob/main/skimage/measure/\\_find\\_contours.py](https://github.com/scikit-image/scikit-image/blob/main/skimage/measure/_find_contours.py)  
findContours.pyx [commit-7511e53]: [https://github.com/scikit-image/scikit-image/blob/main/skimage/measure/\\_find\\_contours\\_cy.pyx](https://github.com/scikit-image/scikit-image/blob/main/skimage/measure/_find_contours_cy.pyx)