Alma Mater Studiorum · Università di Bologna

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

PARALLELIZZAZIONE SU GPU ALGORITMO MARCHING SQUARES PER APPLICAZIONE INDUSTRIALE

Elaborato in:
High Performance Computing

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Correlatore:
Dott.
Matteo Roffilli

Presentata da:
Alessandro Sciarrillo

Anno Accademico 2022/2023

Indice

Capitolo 1

Abstract

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

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

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

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

Capitolo 2

Introduzione alla programmazione Parallela

Capitolo 3

Introduzione e Analisi del Problema

3.1 Introduzione

3.1.1 Marching Squares (MS)

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

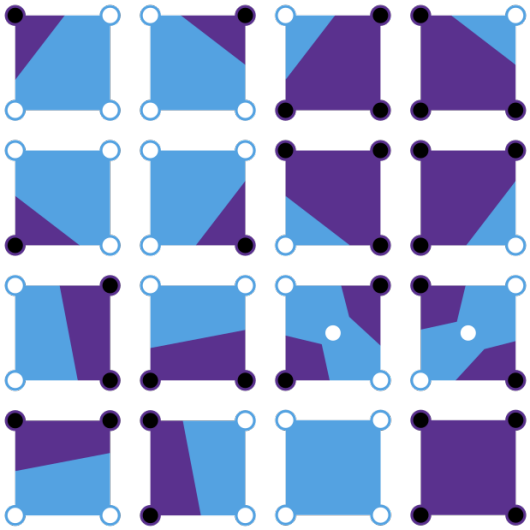

Figura 3.1: Sedici casi possibili in cui possono ricadere i quadrati composti dai quattro valori.

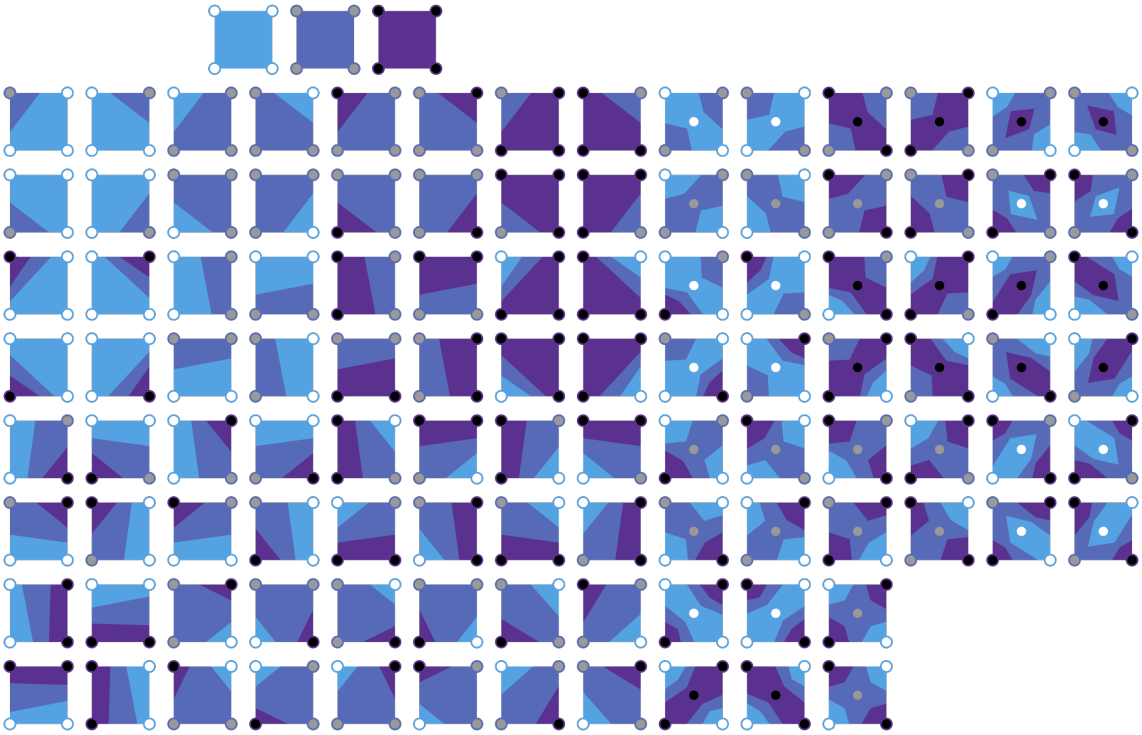

Figura 3.2: Casi possibili nella versione con isobande in cui possono ricadere i quadrati composti dai quattro valori.

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

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

3.1.2 Problema Reale

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

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

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

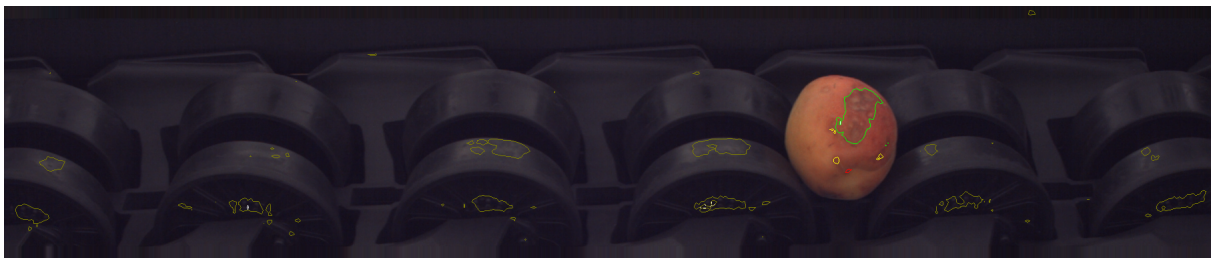

Figura 3.3: Immagine scattata da macchina per la selezione della frutta senza elaborazioni applicate.

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

3.1.3 Obbiettivo

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

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

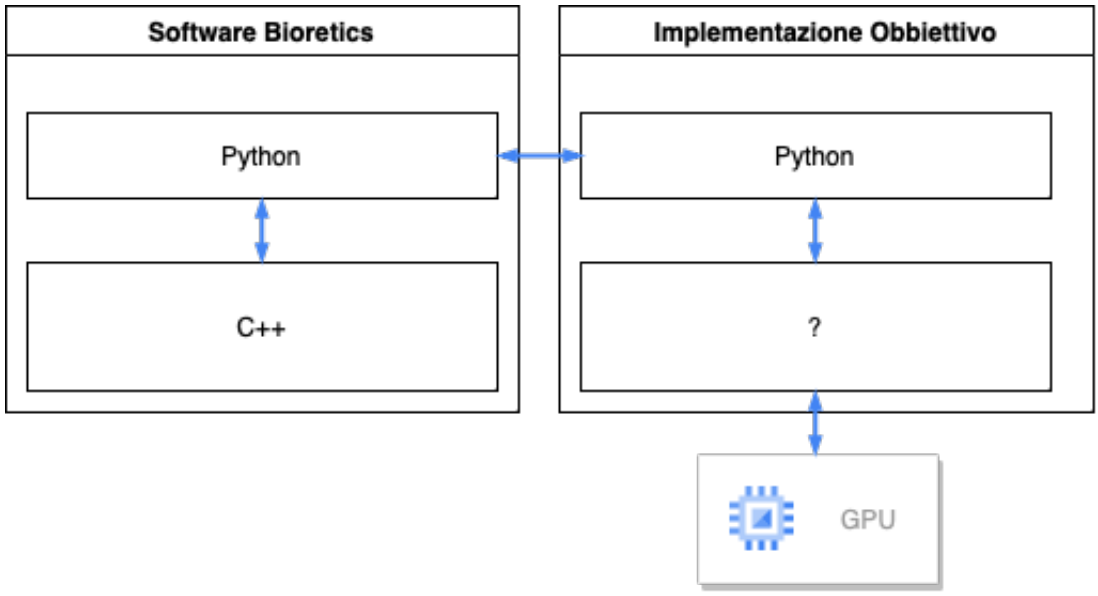

Figura 3.4: Schema linguaggi utilizzati dal software dell'azienda e da utilizzate per la soluzione finale.

3.2 Analisi implementazione skimage

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

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

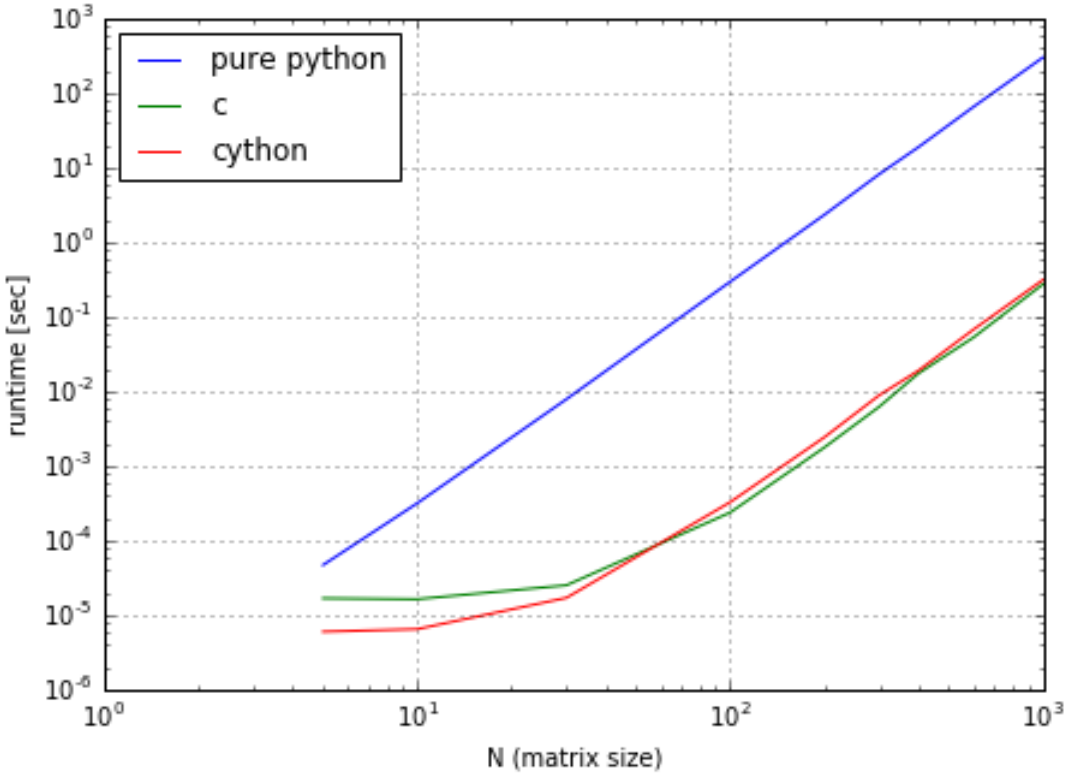

Figura 3.5: Grafico che mette in comparazione i tempi di esecuzione di Pyhon, C e Cython su un task di esempio.

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

3.3 Limiti della programmazione parallela

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

3.4 Analisi versione seriale di MS

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

Codice 1: Metodo find_contours di scikit-image

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

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

Codice 2: Estratto dal Metodo _get_contour_segments di scikit-image

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

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

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

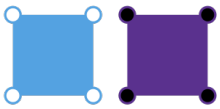

Figura 3.6: Caso 0 e 15 dell'algoritmo Marching Squares.

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

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

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

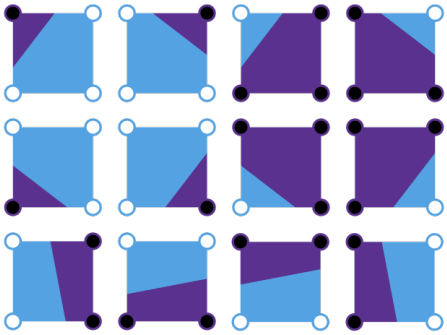

Figura 3.8: Casi 1-5,7,8,10-14 dell'algoritmo Marching Squares.

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

Codice 3: Estratto dal Metodo _get_contour_segments di scikit-image

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

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

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

Estratto dal Metodo _get_contour_segments di scikit-image

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

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

3.4.1 Predisposizione alla parallelizzazione

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

Capitolo 4

Versione parallela con nvc++

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

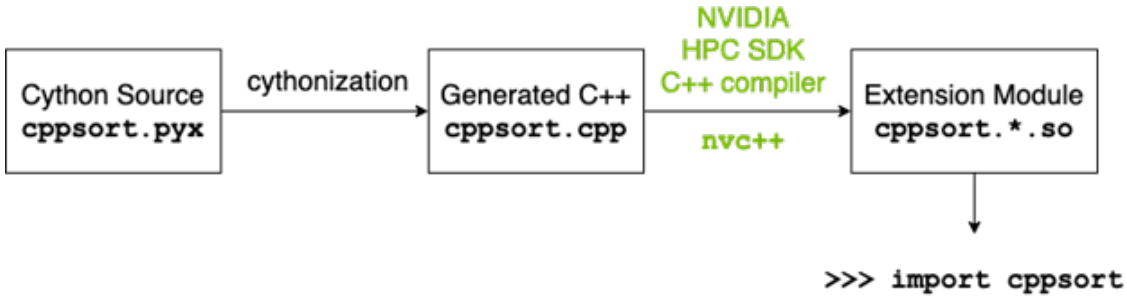

Figura 4.1: Schema di descrizione processo per utilizzo nvc++ in parallelizzazione codice Cython prodotto a Nvidia.

4.1 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, 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 da utilizzare 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 permettesse 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 4.2.

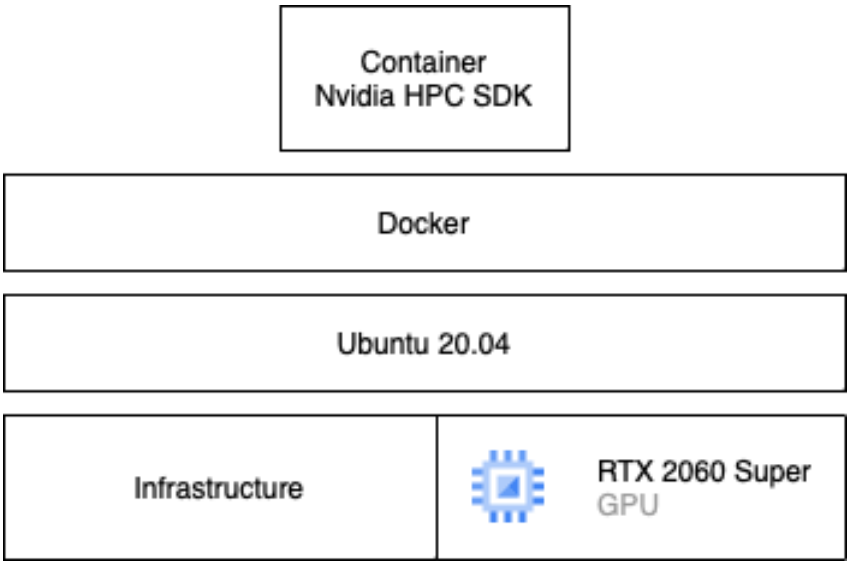

Figura 4.2: Schema componenti hardware e software sul server dell'azienda in seguito alla configurazione del container Nvidia HPC SDK.

4.2 Potenziali risultati

I risultati promessi da Nvidia con l'utilizzo di nvc++ e `-stdpar` sembrano avere un potenziale altissimo, ma vanno ben contestualizzati ai task eseguiti e all'hardware utilizzato.
I benchmarks effettuati dal team developer 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 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 sull'ordinamento di numeri presentati in figura 4.3 è rappresentato lo speedup delle 3 versioni rispetto all'implementazione di Numpy della funzione .sort(). Si può notare uno speedup di circa 20x nel test con il numero maggiore di elementi della versione parallelizzata su GPU da nvc++ con -stdpar rispetto al .sort() di Numpy, risultato apparentemente molto promettente. Nei test con un basso numero di elementi risultano però migliori le versioni su CPU seriali e parallele, fenomeno dovuto all'overhead introdotto da un'esecuzione parallela su GPU che comporta il trasferimento di dati sulle memorie, questo problema è lo stesso discusso in precedenza in relazione alle dimensioni ridotte delle immagini utilizzate nell'applicazione reale oggetto di questa ricerca.

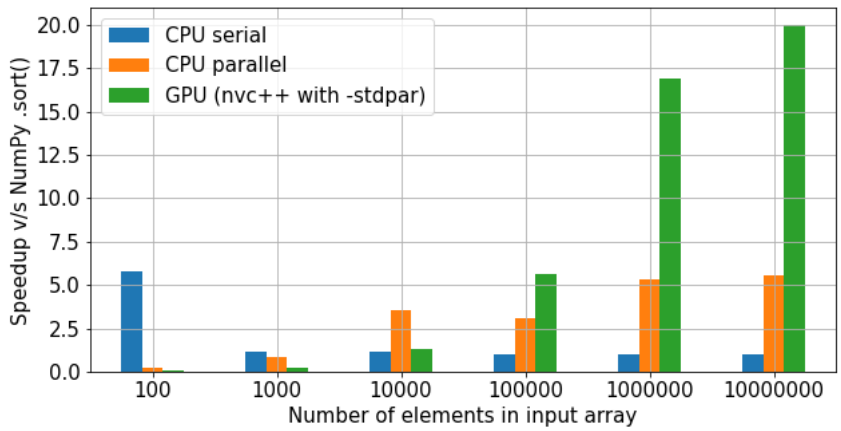

Figura 4.3: Speedup ottenuto a confronto con Numpy nell'ordinamento di una sequenza di interi. I benchmarks su GPU sono stati eseguiti su un sistema con Intel Xeon Gold 6128 CPU, quelli su GPU invece su una NVIDIA A100.

Nei test sulle iterazione del metodo di Jacobi il fenomeno riscontrato con .sort() non si presenta come osservabile nella figura 4.4, il test con il numero di elementi minore è di dimensioni nettamente maggiori 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 benchmarks 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 permettono di avere una potenza di calcolo che è estremamente lontana da quella della maggior parte delle macchine utilizzate in ambito enterprise. Infatti la GPU utilizzata è una Nvidia A100 ovvero una scheda che si trova in configurazioni con decine di GB di memoria a costi di anche 10 volte maggiori a quelli di una 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.

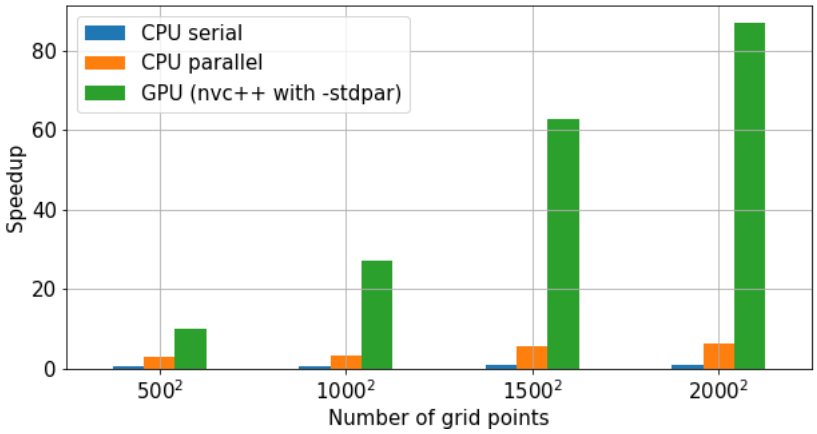

Figura 4.4: Speedup ottenuto a confronto con Numpy in iterazione di Jacobiano. I benchmarks su GPU sono stati eseguiti su un sistema con Intel Xeon Gold 6128 CPU, quelli su GPU invece su una NVIDIA A100.

4.3 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 di skimage in cui è implementato MS.
La strategia scelta per arrivare a questo risultato è quella di partire da una versione di codice basica e poi evolverla in quella finale a piccoli step in modo da risolvere in modo incrementale i problemi che inevitabilmente si presentano in implementazioni di questo tipo dove sono compresi codice di linguaggi diversi e compilatori complessi.
I risultati ottenuti dal team developer di Nvidia si sono subito rivelati un lontano traguardo da raggiungere con hardware di fascia media infatti testando lo stesso codice con cui erano stati effettuati i benchmarks sono stati ottenuti risultati che non erano neanche simili ma anzi peggiori delle versioni seriali e parallele su CPU. Ovviamente risulta difficile pensare di 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, questo 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 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 rischiamare il metodo che verrà poi eseguito in parallelo su GPU.

4.4 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 è infatti come 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 sul processo di parallelizzazione su GPU (descritto nel capitolo 5).

Capitolo 5

Versione parallela con API Cuda-Python

5.1 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 Python alle API dell'host CUDA. Il principale obbiettivo è quello di agevolare per i programmatori l'utilizzo di GPU Nvidia da codice Python.

5.1.1 Requisiti Hardware e Software

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 3.8 a 3.11) che dei propri 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 ed essere compatibile con l'utilizzo di CUDA.

5.1.2 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 in PTX (Parallel Thread Execution) ed estrarne la funzione che potrà poi essere richiamata dall'applicazione python. PTX è una macchina virtuale per 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 (codice Cuda che verrà eseguito su GPU) viene scritto in una

stringa python e compilato con NVRTC che è una libreria di compilazione runtime per CUDA C++.
Il processo di esecuzione parallela del codice Cuda partendo da codice Python in generale consiste nell'utilizzo delle API per i 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

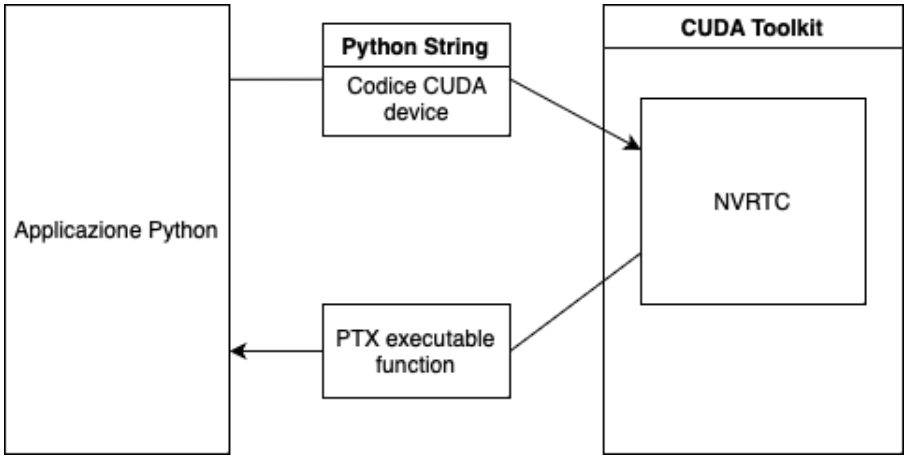

Figura 5.1: Schema compilazione runtime con NVRTC di codice CUDA C++ da codice Python e output funzione compatibile PTX richiamabile da Python.

- caricamento dati da elaborare su GPU
- lancio codice compilato CUDA C++ su GPU
- recupero risultati da GPU

5.1.3 Utilizzo funzioni principali

Per prima cosa vanno importati NVRTC e le API dei Driver dal pacchetto CUDA-Python come riportato nella seguente portzione 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 poi essere richiamata dal Python. In questo codice di esempio viene cercata una capacità di computazione di 75 con FMAD attivato.

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 amcchina alla creazione `context` va passato l'indicatore 0 alla funzione `cuCtxCreate`. 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, un modulo è l'analogo per il device di librerie caricare dinamicamente. Più kernel posso essere contenuti all'interno di PTX, in seguito al carimento 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  ASSERT_DRV(err)
7
8  # Retrieve function
9  err, kernel = cuda.cuModuleGetFunction(module, b"saxpy")
10 ASSERT_DRV(err)
```

In seguito 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  NUM_THREADS = 512  # Threads per block
2  NUM_BLOCKS = 32768  # Blocks per grid
3
4  a = np.array([2.0], dtype=np.float32)
5  n = np.array(NUM_THREADS * NUM_BLOCKS, 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 allocati e caricati i kernel sono pronti per essere lanciati, per farlo è necessario avere a disposizione gli indirizzi dei dati sul device per essere passati alle configurazioni di esecuzione. 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

```
dIn = np.array([int(dInclass)], dtype=np.uint64)
dOut = np.array([int(dOutclass)], dtype=np.uint64)

args = [a, dIn, dOut, n]
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 effettivamente serializzati.

Lancio kernel

```
err, = cuda.cuLaunchKernel(
    kernel,
    NUM_BLOCKS,  # grid x dim
    1,  # grid y dim
    1,  # grid z dim
    NUM_THREADS,  # block x dim
    1,  # block y dim
    1,  # block z dim
    0,  # dynamic shared memory
    stream,  # stream
    args.ctypes.data,  # kernel arguments
    0,  # extra (ignore)
)
```

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, infatti la chiamata verrà risolta solo al termine dell'esecuzione del kernel.
Prima di proseguire con altre chiamate per esempio a nuovi kernel è utilie 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 l allocata su GPU e distruggendo sia lo `stream` che il `contex` 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 ovvero la 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 aggioranta.

5.1.4 Prestazioni dichiarate

Nvidia riporta i benchmark effettuati sul del codice di base ovvero tutte le istruzioni descritte in precedenza per l'utilizzo delle API con un semplice kernel CUDA che effettua semplici operazioni di algebra lineare chiamato SAXPY (Single-Precision A·X Plus Y).
Ogni thread effettua solamente una operazione $a \cdot x + y$ utilizzando un calore 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 [ ] 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 totale che si discostano 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 5.1: Comparazione tempi di esecuzione kernel e applicazione completa tra versione in C++ e Python.

5.2 Progettazione struttura codice parallelo

La soluzione da sviluppare dovrà 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'azieda che tramite le le API dei Driver Cuda riesca a caricare i dati, lanciare i kernel e recuperare il riusltato 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 5.2.

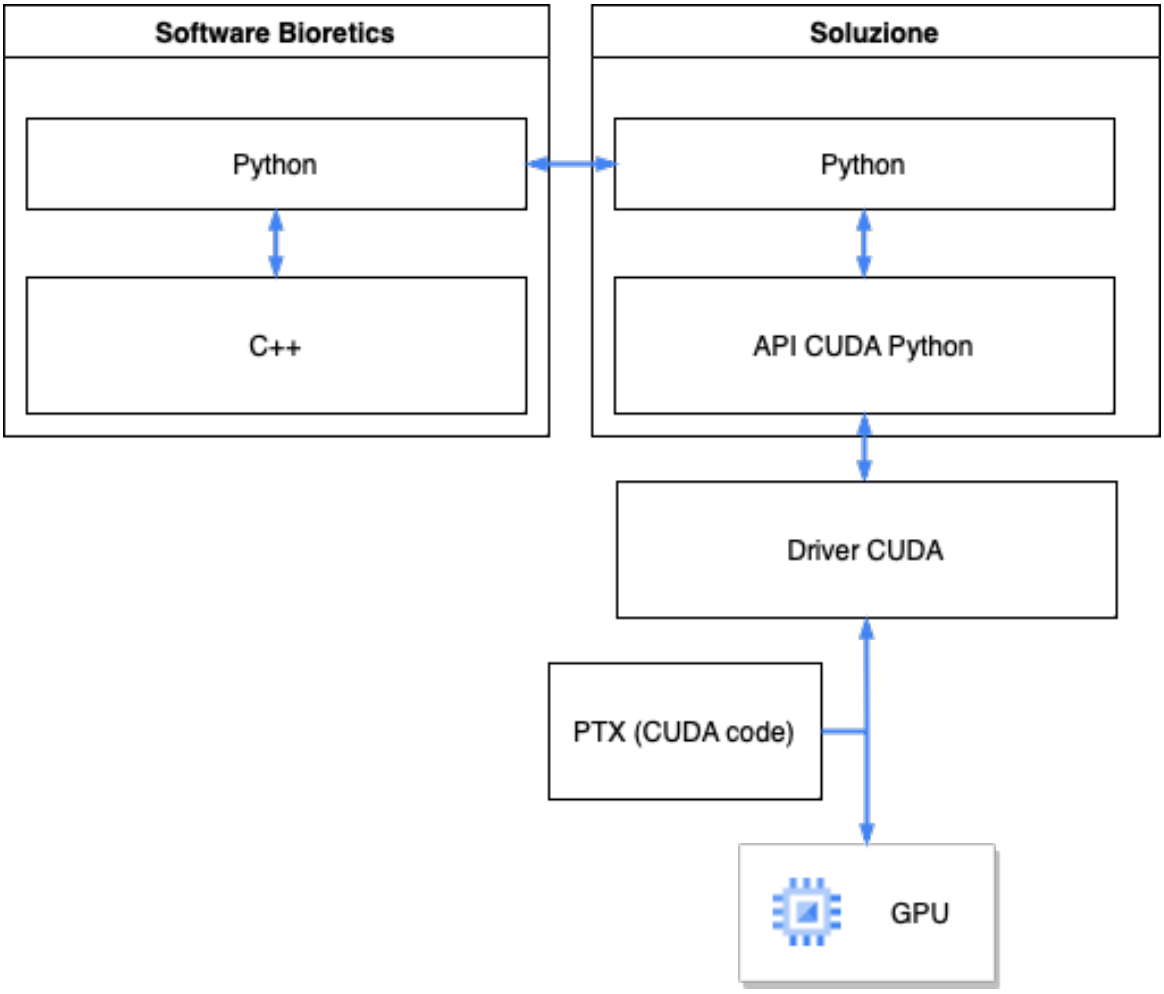

Figura 5.2: Schema utilizzo API CUDA-Python per utilizzo GPU da Python.

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 funzine Python potrà essere importata nel codice dell'azienda ed essere quindi direttamente richiamabile, si occuperà quindi di interfacciarsi con le applicazioni esterno 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, andrà in seguito portato dalla memoria della scheda video a quella dell'host il risultato ottenuto con l'esecuzione parallela.

5.2.1 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 poi 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 contente 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 introdotto del padding tra un elemento e l'altro, nel nostro caso il tipo utilizzato è il `double` che occupa 8 bytes e potrebbero quindi essere probabilmente disposti in sequenza in memoria senza l'aggiunta di padding. Per quanto riguarda una tupla contente due tuple però non è possibile stabilire con certezza come verranno disposti in memoria, non è quindi una soluzione ottimizzata utilizzare un array contentente queste tuple.
Una struttura dati supportata nativamente da C che possa sostituire una lista di tuple contenetei tuple di double sono un inisieme di 4 array, non è la sola a poter esser utilizzata ma è quella che ottimizza maggiormante l'utilizzo dello spazio dei propri elementi, dello spazio totale e all'accesso alle posizioni specifiche. Ovviamente mantenere 4 array disconnessi fisicamente ma uniti solo a livello logico sono molto più delicati e richiedono una attenzione maggiore per essere mantenuti 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 5.3 e quella che verrà utilizzata per la versione parallela nella figura 5.4.

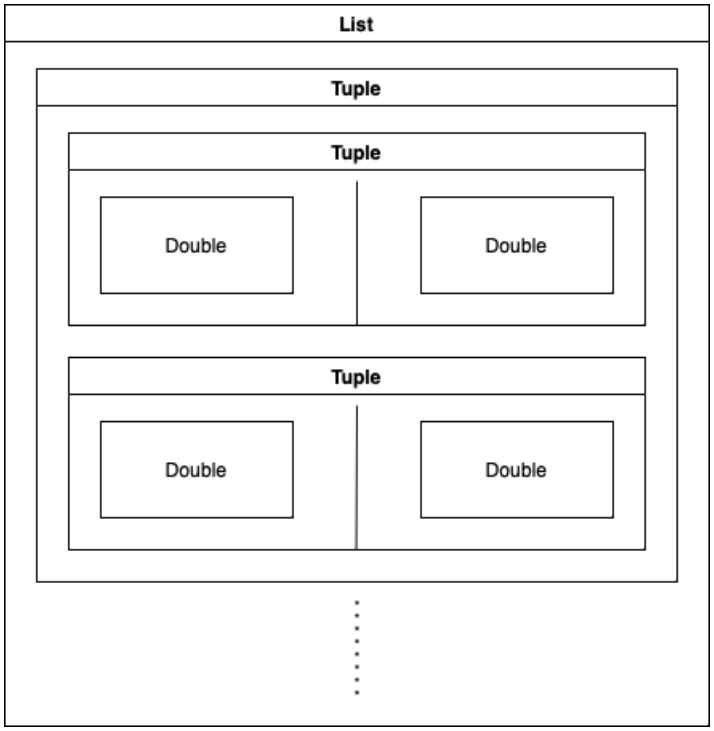

Figura 5.3: Insieme di strutture dati utilizzate per la versione Python di skimage.

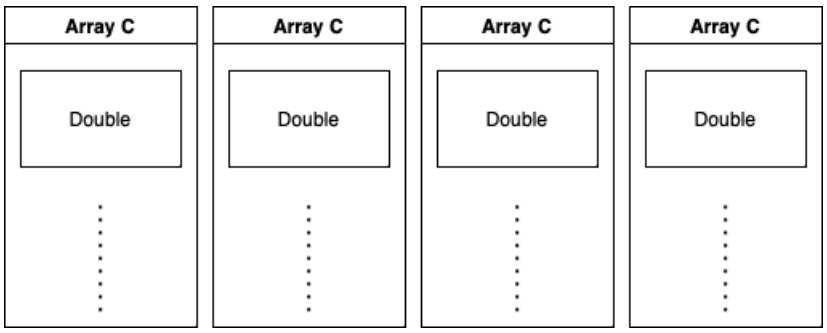

Figura 5.4: Insieme di strutture dati utilizzate per la versione parallela.

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 che sfruttano interamente i bus invece che solo parzialmente ricorrendo in una penalizzazione sui tempi.

5.3 Progettazione kernel Cuda

La fase di progettazione dei kernel Cuda è quella che ha richiesto il maggior studio e pianificazione in quanto la parallelizzazione 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 sarà sufficiente ma saranno necessari diversi kernel da lanciare in sequenza per riuscire a replicare il funzionamento del codice seriale in parallelo.

5.3.1 Kernel required_memory

Dato che per utilizzare gli array è necessario definire al momento della loro creazione la dimensione che occuperanno in memoria è necessario stabilire quante posizioni degli array 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 quaderato composto da 4 valori confinanti verranno stabilite quante posizioni dell'array occuperanno.

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 dei quadrati analizzabili dell'immagine che conterrà nella posizione rispettiva all'immagine un valore

da 0 a 2 che indica quanti segmenti sarà necessario memorizzare per quel preciso quadrato. come mostrato nella figura 5.5.

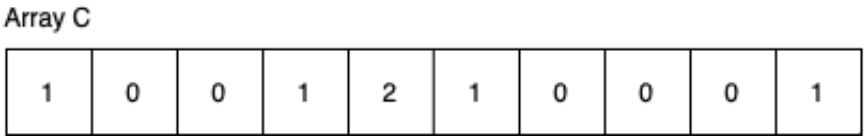

Figura 5.5: Array C risultato dell'esecuzione del kernel required_memory.

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 in relazione alla versione parallela estrememente lenta. Sono quindi necessarie altre operazioni parallele per poter ottenere la somma di tutti gli elementi dell'array generato da questo primo kernel.

5.3.2 Kernel reduce

Per calcolare la somma di tutti gli elementi dell'array output del kernel required_memory è necessario effettuare una operazione di riduzione che è possibile parallelizzare.
La reduce viene effettuata utilizzando ad ogni passaggio un numero di Cuda Core pari alla metà del numero di elementi rimasti, ad ogni iterazione ogni Cuda Core 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 5.6.

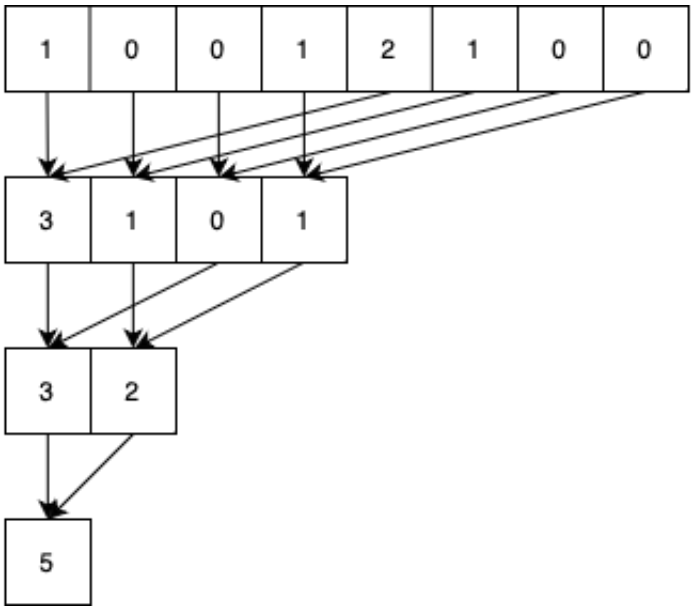

Figura 5.6: Operazione di reduce.

Ad ogni passo si dimezzano come il numero di elementi restanti anche il numero di thread utilizzati, questo significa che ci saranno thread che resteranno senza alcun lavoro da svolgere. Una accortezza è quella di lanciare il kernel con un numero di thread che siano 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 incosistenti 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 nella locazione di memoria del risultato dal thread 0. Ogni blocco di thread ottiene quindi la somma dei valori corrispondenti alle loro posizioni, una volta riportati questi risultati parziali sull'host è necessario effettuare un'ulteriore operazione di somma su di questi per ottenere il risultato finale. In questo caso però il numero di blocchi 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 rispetto a effetture la somma degli elementi 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()
```

5.3.3 Kernel exclusive scan (prescan)

Una volta ricavato quanto sia il totale di segmenti che andranno memorizzati manca ancora un 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 riservata ognuno di loro dovrà scrivere.
Esiste una funzione da applicare ad 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, calcola tutti i prefissi di un array in base ad una operazione data che nel nostro caso sarà l'operatore binario di somma.
La exclusive scan è caratterizzata dal fatto che nella prima posizione dell'array risultato venga messo il valore dell'elemento neutrale dell'operazione utilizzata che è 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 di una exclusive scan è riportato nella figura 5.7.

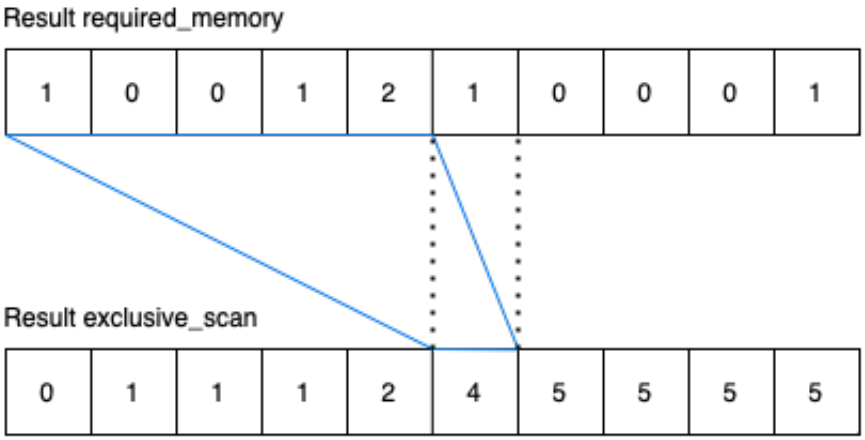

Figura 5.7: Vettori di input e output della funzione di exclusive scan.

Il risultato di una exclusive scan su un array per le sue proprietà è già pronto per essere utilizzato per lo scopo desiderato infatti in ogni posizione è contenuta la posizione in cui andrà scritto il rispettivo valore. Rispetta anche gli indici degli array infatti la prima posizione indicata in cui scrivere è lo 0, per quelle successive invece il valore della posizione non considera quanti segmenti andranno scritti da quel thread ma solo quelli precedenti che è corretto dato che è necessario saper solo da dove iniziare a scrivere. Sono presenti posizioni ripetute nel vettore in output, il risultato è corretto in quanto l'indice della prossima posizione in cui scrivere non deve variare se i rispettivi thread non hanno alcun segmento da scrivere. I thread che invece dovranno scrivere 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. Sono necessari infatti un totale di 3 kernel differenti per comporre il vettore finale in modalità parallela:

- prescan (exclusive scan)
- prescan_small (exclusive scan small)
- add

Non è possibile effettuare tutti i passaggi necessari in un unico kernel poichè ...

kernel prescan

kernel prescan_small

kernel add

5.4 Implementazione

5.4.1 Implementazione modulo python

5.4.2 Implementazione kernel

5.5 Risultati ottenuti

Capitolo 6

Risultati a Confronto

Capitolo 7

Conclusioni