UNIVERSITÀ DEGLI STUDI DI NAPOLI
"PARTHENOPE"

SCUOLA INTERDIPARTIMENTALE DELLE SCIENZE,
DELL'INGEGNERIA E DELLA SALUTE
Dipartimento di Scienze e Tecnologie

Corso di Laurea Triennale in Informatica

UN APPROCCIO PARALLELO SU GPU PER SIMULAZIONI SU LARGA SCALA DELL'ANGIOGENESI

A GPU-PARALLEL APPROACH FOR LARGE-SCALE ANGIOGENESIS SIMULATIONS

Relatore:

Prof. Livia Marcellino

Candidato:

Anna Greco
Matr. 0124002362

Correlatore:

Prof. Pasquale De Luca

ANNO ACCADEMICO 2023/2024

Abstract

La simulazione computazionale rappresenta un metodo essenziale per lo studio di fenomeni biologici complessi, permettendo di esplorare scenari difficilmente accessibili attraverso esperimenti tradizionali. Questa ricerca si concentra sull'implementazione e ottimizzazione di un modello computazionale per l'angiogenesi, un processo fondamentale sia per la rigenerazione e crescita dei tessuti, sia per lo sviluppo di patologie, come il cancro. Utilizzando il Modello Cellulare di Potts (CPM), questo studio indaga la migrazione e l'interazione cellulare in risposta a stimoli chimici e campi elettrici.

L'approccio iniziale, sviluppato in forma sequenziale, è stato completamente riprogettato per sfruttare la parallelizzazione su GPU tramite la piattaforma CUDA di NVIDIA. Nello specifico, attraverso l'ottimizzazione delle routine computazionali e l'integrazione di librerie avanzate, tra cui cuDSS e CUB, la versione parallela ha raggiunto prestazioni significativamente superiori, riducendo drasticamente i tempi di simulazione e consentendo l'analisi di sistemi biologici su larga scala, superando le limitazioni computazionali precedenti.

Il lavoro applica inoltre estensioni al CPM, incorporando termini per descrivere fenomeni quali la chemiotassi e l'interazione con campi elettrici, ampliando così le capacità del modello nella rappresentazione di processi biologici complessi. I risultati evidenziano non solo l'efficacia della parallelizzazione, ma anche il valore dell'integrazione di approcci computazionali avanzati nell'affrontare alcuni aspetti delle sfide scientifiche contemporanee.

In conclusione, questo studio si propone come un contributo al campo della biologia computazionale, mettendo a disposizione uno strumento scalabile e innovativo, pensato per supportare la ricerca scientifica di fenomeni biologici complessi.

Abstract (English)

Computational simulation is an essential method for studying complex biological phenomena, enabling the exploration of scenarios that are challenging to investigate experimentally. This research focuses on the implementation and optimization of a computational model for angiogenesis, a crucial process for tissue regeneration and growth and for the development of pathologies such as cancer. Using the Cellular Potts Model (CPM), this study examines cell migration and interaction in response to chemical stimuli and electric fields.

The initial sequential approach was fully redesigned to leverage GPU parallelization through NVIDIA's CUDA platform. By optimizing computational routines and integrating advanced libraries such as cuDSS and CUB, the parallel version achieved significantly improved performance, drastically reducing simulation times and enabling the analysis of large-scale biological systems, overcoming previous computational limitations.

The study also extends the CPM by incorporating terms to describe phenomena such as chemotaxis and interactions with electric fields, thereby enhancing the model's ability to represent complex biological processes. The findings highlight the effectiveness of parallelization and the importance of integrating advanced computational methods to address some aspects of modern scientific challenges.

In conclusion, this work aims to contribute to computational biology by proposing an innovative and scalable tool to support scientific research on complex biological phenomena.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli algoritmi

Introduzione

Negli ultimi anni, le simulazioni computazionali hanno assunto un ruolo centrale in molteplici ambiti scientifici, permettendo di modellare fenomeni complessi e di esplorare scenari difficilmente accessibili con esperimenti tradizionali. In particolare, lo studio dell'*angiogenesi*, il processo biologico che porta alla formazione di nuovi vasi sanguigni, rappresenta una sfida rilevante sia in ambito biologico che clinico. Comprendere le dinamiche che regolano questo processo è fondamentale per sviluppare interventi efficaci in condizioni patologiche, come il *cancro*, dove l'angiogenesi contribuisce alla progressione tumorale.

Questo lavoro si propone di affrontare il problema della simulazione su larga scala dell'angiogenesi, utilizzando un approccio basato sul *Modello Cellulare di Potts* (CPM). Dopo un'implementazione iniziale del modello in linguaggi sequenziali come *MATLAB* e *C*, è stata sviluppata una versione parallela ottimizzata sfruttando le *GPU* e la piattaforma *CUDA*. L'ottimizzazione delle routine computazionali e l'integrazione di librerie avanzate hanno reso il codice parallelo estremamente efficiente, consentendo di ridurre drasticamente i tempi di esecuzione e di simulare scenari biologici realistici e complessi su vasta scala.

Il lavoro si articola in una fase iniziale di analisi delle basi biologiche e computazionali del modello, seguita da un'implementazione progressiva che parte dal codice sequenziale per arrivare alla sua versione parallela. I risultati ottenuti dimostrano l'elevata efficienza del codice parallelo, capace di scalare in modo ottimale con l'aumentare delle dimensioni del problema, garantendo prestazioni computazionali di alto livello.

Questo studio si pone l'obiettivo di fornire una base per futuri sviluppi, come l'integrazione di tecniche di apprendimento automatico per l'analisi predittiva o l'estensione a modelli tridimensionali. Al fine di offrire uno strumento innovativo per la ricerca medica, con potenziali applicazioni rilevanti nella diagnosi e nel trattamento di patologie complesse.

Obiettivi del lavoro

Il focus principale di questo studio è lo sviluppo di un approccio parallelo per simulazioni su larga scala del processo di angiogenesi utilizzando GPU. In particolare, il lavoro si concentra sull'implementazione e sull'ottimizzazione del Modello Cellulare di Potts (CPM), estendendo il modello con termini aggiuntivi per includere la *chemiotassi* e il campo elettrico, al fine di migliorare la rappresentazione della dinamica cellulare. Lo scopo è duplice: da un lato, comprendere meglio le dinamiche di angiogenesi attraverso simulazioni realistiche; dall'altro, proporre soluzioni computazionali che sfruttino l'efficienza delle GPU per ridurre i tempi di simulazione.

Motivazione e contesto della ricerca

L'angiogenesi, gioca un ruolo centrale sia in condizioni fisiologiche che patologiche. Comprendere questo fenomeno è fondamentale per sviluppare nuove strategie terapeutiche, in particolare per malattie come il cancro, dove l'angiogenesi contribuisce alla progressione tumorale. La simulazione dell'angiogenesi offre un potente strumento per analizzare le dinamiche molecolari e cellulari che regolano questo processo, consentendo di testare virtualmente nuove terapie.

La crescente disponibilità di GPU ha aperto nuove opportunità per simulazioni su larga scala, rendendo possibile l'esecuzione di modelli computazionali complessi in tempi ragionevoli. La presente ricerca si inserisce in un contesto in cui l'efficienza computazionale e la capacità di rappresentare accuratamente fenomeni biologici complessi sono fondamentali per avanzare nella ricerca scientifica e clinica.

Breve panoramica sull'angiogenesi e la rilevanza delle simulazioni

L'angiogenesi è un processo biologico fondamentale che regola la formazione di nuovi vasi sanguigni a partire da quelli preesistenti. Questo meccanismo è essenziale per una vasta gamma di fenomeni fisiologici, come la riparazione dei tessuti, la crescita embrionale e la rigenerazione cellulare. Tuttavia, in contesti patologici, l'angiogenesi può risultare deregolata, contribuendo allo sviluppo di malattie come il cancro, la retinopatia diabetica e le malattie cardiovascolari. In particolare, nei tumori solidi, l'angiogenesi è uno dei principali meccanismi che alimentano la crescita tumorale e favoriscono la formazione di metastasi, rappresentando un obiettivo chiave per lo sviluppo di terapie innovative.

Questo processo è regolato da un delicato equilibrio tra segnali pro-angiogenici, come il VEGF (*Vascular Endothelial Growth Factor*), e segnali anti-angiogenici, che insieme orchestrano la migrazione e la proliferazione delle cellule endoteliali. L'interazione tra fattori biochimici, gradienti chimici e segnali meccanici gioca un ruolo centrale nella dinamica dell'angiogenesi, rendendo la sua modellazione computazionale particolarmente complessa e multidimensionale.

Le simulazioni computazionali rappresentano uno strumento prezioso per investigare questi fenomeni. Esse permettono di esplorare come le cellule rispondano a fattori esterni, come gradienti chimici, campi elettrici e interazioni cellula-cellula, offrendo una prospettiva unica per comprendere i meccanismi sottostanti e testare nuove strategie terapeutiche. Inoltre, grazie all'elevata precisione delle simulazioni, è possibile modellare scenari difficilmente accessibili sperimentalmente, riducendo al contempo i costi e i tempi associati alla ricerca tradizionale.

Tra i diversi approcci computazionali, il Modello Cellulare di Potts si è dimostrato particolarmente efficace per studiare fenomeni come la migrazione cellulare, l'adesione e le interazioni con il microambiente. Questo modello consente di rappresentare le cellule come entità discrete che interagiscono attraverso parametri energetici, facilitando la simulazione delle loro dinamiche. L'estensione del CPM per includere processi come la chemiotassi (migrazione guidata da gradienti chimici) e la risposta ai campi elettrici offre la possibilità di esplorare fenomeni biologici complessi con un elevato grado di dettaglio.

In particolare, la combinazione di gradienti chimici e campi elettrici consente di modellare il comportamento delle cellule endoteliali in condizioni sia fisiologiche che patologiche, fornendo informazioni preziose sulle dinamiche dell'angiogenesi.

Queste simulazioni non solo migliorano la nostra comprensione teorica del processo, ma offrono anche un potenziale strumento predittivo per la ricerca medica, con implicazioni nella diagnosi precoce e nello sviluppo di trattamenti personalizzati.

Struttura della tesi

Questa tesi è strutturata in diversi capitoli, ciascuno dei quali affronta un aspetto chiave del lavoro:

1. **Contesto scientifico e tecnologico**:

   Analizza i principi biologici dell'angiogenesi, il CPM e le tecniche di parallelizzazione con GPU, evidenziando l'importanza di queste metodologie nella simulazione su larga scala.

2. **Metodologia: dal sequenziale al parallelo**:

   Illustra il processo di sviluppo del codice, partendo dall'implementazione sequenziale in MATLAB, passando per l'ottimizzazione in C e concludendo con la parallelizzazione in CUDA.

3. **Architetture parallele e programmazione con CUDA**:

   Approfondisce i dettagli tecnici dell'implementazione parallela, analizzando le sfide e le soluzioni adottate.

4. **Dall'implementazione parallela ai risultati**:

   Confronta le performance tra codice sequenziale e parallelo, valutando l'accuratezza dei risultati e identificando le limitazioni attuali. Si conclude con una riflessione sulle possibili direzioni per il miglioramento e l'estensione del lavoro.

Capitolo 1

Contesto Scientifico e Tecnologico

1.1 Angiogenesi e modellizzazione biologica

1.1.1 Introduzione all'angiogenesi

L'angiogenesi è il processo mediante il quale nuovi vasi sanguigni si formano a partire da vasi preesistenti, giocando un ruolo fondamentale nella crescita e nel mantenimento dei tessuti.

Questo fenomeno è essenziale non solo per lo sviluppo normale dell'organismo, come durante la crescita embrionale e la riparazione delle ferite, ma anche per l'adattamento dei tessuti a cambiamenti come l'esercizio fisico intenso.

L'angiogenesi è inoltre implicata in una serie di condizioni patologiche, come il cancro, dove l'apporto di nuovi vasi sanguigni consente al cancro di ottenere ossigeno e nutrienti necessari per crescere e diffondersi in altre parti del corpo [21] *(Hanahan & Weinberg, 2011)*.

Il processo angiogenico è guidato da un complesso sistema di segnali molecolari che includono sia fattori promotori che inibitori. I principali componenti di questo processo sono le cellule endoteliali, che rivestono l'interno dei vasi sanguigni. Queste cellule rispondono a segnali angiogenici, come il Fattore di Crescita Endoteliale Vascolare (VEGF), attivandosi, proliferando e migrando verso aree in cui è necessario sviluppare nuovi vasi. La regolazione dell'angiogenesi dipende da un equilibrio dinamico tra fattori pro-angiogenici, come VEGF e Fattori di Crescita dei Fibroblasti (FGF), e inibitori naturali come l'angiostatina e l'endostatina [5] *(Carmeliet, 2005)*.

Dal punto di vista fisiologico, l'angiogenesi è necessaria per processi come la rigenerazione dei tessuti e la cicatrizzazione delle ferite, mentre in un contesto patologico essa può essere coinvolta in condizioni quali la retinopatia diabetica, la degenerazione maculare senile e la progressione tumorale.

La capacità del cancro di indurre l'angiogenesi, spesso denominata "*switch angiogenico*", è un elemento critico che consente al cancro di superare la fase di crescita limitata dalla mancanza di apporto sanguigno e di progredire verso uno stato maligno e invasivo [25] *(Jain, 2005)*.

La comprensione approfondita delle differenze tra angiogenesi fisiologica e patologica rappresenta un passo fondamentale per sviluppare strategie terapeutiche che mirino selettivamente a inibire la crescita dei vasi nei contesti patologici, preservando al contempo le normali funzioni del sistema vascolare.

1.1.2 Ruolo chiave dell'angiogenesi nelle condizioni fisiologiche e patologiche

L'angiogenesi si manifesta sia in condizioni fisiologiche, dove contribuisce alla crescita e alla rigenerazione dei tessuti, sia in contesti patologici, dove la sua regolazione può portare a complicazioni significative.

In *condizioni fisiologiche*, l'angiogenesi è fondamentale per il corretto sviluppo embrionale, per la formazione della placenta durante la gravidanza, e per la riparazione dei tessuti danneggiati, come avviene durante la guarigione delle ferite o la risposta a esercizi fisici intensi [5] *(Carmeliet, 2005)*. Questi processi sono caratterizzati da una stretta regolazione da parte di fattori pro-angiogenici e inibitori che mantengono l'equilibrio tra crescita e inibizione vascolare, assicurando la stabilità e funzionalità del sistema vascolare.

In *condizioni patologiche*, invece, l'angiogenesi può essere eccessivamente stimolata o deregolata, portando a problemi significativi. Nel cancro, ad esempio, l'angiogenesi patologica favorisce la crescita tumorale e la formazione di metastasi, poiché consente al cancro di ricevere nutrienti e ossigeno attraverso nuovi vasi sanguigni [21] *(Hanahan & Weinberg, 2011)*.

Il cancro può indurre l'angiogenesi attraverso la secrezione di fattori di crescita, come il VEGF, che stimolano le cellule endoteliali a proliferare e a formare nuovi vasi in grado di alimentare la massa tumorale in crescita [25] *(Jain, 2005)*.

Allo stesso modo, patologie come la retinopatia diabetica e la degenerazione maculare senile sono caratterizzate da una crescita anomala dei vasi sanguigni, che può compromettere gravemente la funzione dei tessuti coinvolti.

La capacità di modulare l'angiogenesi, favorendola nei contesti in cui è necessaria (come nella riparazione dei tessuti) e inibendola nei casi in cui è dannosa (come nei tumori), rappresenta una delle principali sfide della ricerca medica moderna [16] *(Ferrara & Kerbel, 2005)*. Terapie anti-angiogeniche, basate sull'inibizione del VEGF e di altri fattori pro-angiogenici, sono attualmente impiegate in ambito oncologico per ridurre l'apporto di sangue ai tumori, ostacolandone così la crescita e la diffusione [16] *(Ferrara & Kerbel, 2005)*.

1.1.3 Fattori chiave dell'angiogenesi

Il processo di angiogenesi è regolato da una serie di fattori chiave che orchestrano la proliferazione, la migrazione e la differenziazione delle cellule coinvolte. Tra i principali protagonisti dell'angiogenesi troviamo le cellule endoteliali, il Fattore di Crescita Endoteliale Vascolare (VEGF), le metalloproteasi della matrice (MMPs), e la matrice extracellulare (ECM). Le cellule endoteliali sono responsabili dell'inizio e del mantenimento del processo angiogenico; in risposta ai segnali angiogenici, queste cellule degradano la membrana basale circostante, proliferano e migrano per formare nuovi vasi sanguigni [5] *(Carmeliet, 2005)*.

Il *VEGF* è uno dei più importanti fattori pro-angiogenici. Questo fattore di crescita, spesso definito come il "regolatore principale" dell'angiogenesi, stimola la proliferazione e la migrazione delle cellule endoteliali e gioca un ruolo critico nella formazione di nuovi vasi.

Il VEGF è secreto in risposta a stimoli come l'ipossia, e la sua attività è fondamentale per la crescita tumorale, poiché fornisce ai tumori il supporto vascolare di cui hanno bisogno per crescere e diffondersi [21] *(Hanahan & Weinberg, 2011)*.

Le *metalloproteasi della matrice* (MMPs) sono enzimi che degradano la matrice extracellulare, un processo necessario per consentire la migrazione delle cellule endoteliali e creare spazio per la formazione di nuovi vasi. La regolazione delle MMPs è fondamentale per garantire che la degradazione della matrice avvenga in modo controllato, prevenendo eccessi che potrebbero portare a danni tissutali [25] *(Jain, 2005)*.

La *matrice extracellulare* (ECM) fornisce supporto strutturale alle cellule e regola la loro migrazione e proliferazione durante l'angiogenesi. La ECM non solo agisce come impalcatura per le cellule endoteliali, ma fornisce anche segnali biochimici che influenzano il comportamento cellulare. Durante l'angiogenesi, la ECM viene continuamente rimodellata per consentire l'espansione della rete vascolare e la stabilizzazione dei nuovi vasi formati [5] *(Carmeliet, 2005)*.

Il coordinamento tra questi fattori è essenziale per garantire un processo angiogenico efficace e ben regolato, che sia capace di rispondere adeguatamente alle necessità dell'organismo sia in condizioni fisiologiche che patologiche.

1.1.4 Il processo di angiogenesi

Il processo di angiogenesi comprende diverse fasi chiave, ognuna delle quali è essenziale per la formazione di nuovi vasi sanguigni. Inizialmente, l'angiogenesi è innescata da uno stimolo angiogenico, come l'ipossia, che induce il rilascio di fattori di crescita come il Fattore di Crescita Endoteliale Vascolare (VEGF) [21] *(Hanahan & Weinberg, 2011)*. Questi fattori agiscono sulle cellule endoteliali presenti nei vasi esistenti, inducendo la loro attivazione, proliferazione e migrazione.

La degradazione della membrana basale è uno dei primi passi dell'angiogenesi. In risposta a segnali angiogenici, le cellule endoteliali secernono enzimi chiamati metalloproteasi della matrice (MMPs), che degradano la membrana basale e la matrice extracellulare circostante, permettendo alle cellule di migrare verso il tessuto bersaglio [25] *(Jain, 2005)*.

Successivamente, le cellule endoteliali iniziano a migrare verso la sorgente del segnale angiogenico. La migrazione delle cellule endoteliali è guidata da segnali chemiotattici, come il VEGF, che funge da attrattore per queste cellule. La proliferazione delle cellule endoteliali permette poi la formazione di un cordone di cellule che andrà a costituire il nuovo vaso [5] *(Carmeliet, 2005)*.

Una volta che le cellule endoteliali hanno formato un cordone, si verifica la formazione del lume, un processo mediante il quale le cellule endoteliali riorganizzano la loro struttura per creare un canale attraverso cui il sangue può fluire. Durante questa fase, il vaso neonato si connette con i vasi preesistenti attraverso un processo chiamato anastomosi, che garantisce l'integrazione del nuovo vaso nella rete vascolare già presente.

Infine, il nuovo vaso sanguigno deve essere stabilizzato e maturato. Questa fase richiede il reclutamento di cellule di supporto, come i periciti e le cellule muscolari lisce, che contribuiscono alla stabilità del vaso e al mantenimento della sua funzionalità [45] *(Ribatti, 2007)*. La membrana basale viene anche riformata per fornire ulteriore supporto strutturale e protezione al nuovo vaso sanguigno.

Il processo di angiogenesi è strettamente regolato da un equilibrio tra segnali pro-angiogenici e anti-angiogenici. Quando questo equilibrio è disturbato, si possono verificare patologie, come la crescita tumorale o la degenerazione di tessuti, evidenziando l'importanza di una corretta regolazione dell'angiogenesi in contesti sia fisiologici che patologici [16] *(Ferrara & Kerbel, 2005)*.

1.1.5 Angiogenesi e cancro

L'angiogenesi gioca un ruolo fondamentale nello sviluppo e nella progressione del cancro. La capacità di un cancro di crescere oltre una certa dimensione e di formare metastasi dipende in gran parte dalla sua abilità di stimolare la formazione di nuovi vasi sanguigni per garantire l'apporto di ossigeno e nutrienti [21] *(Hanahan & Weinberg, 2011)*. Questo fenomeno, noto come switch angiogenico, rappresenta una tappa critica nella progressione del cancro da uno stato avascolare e limitato a uno stato maligno e invasivo.

La transizione verso uno stato pro-angiogenico è guidata da uno squilibrio tra fattori pro-angiogenici e anti-angiogenici. I tumori secernono elevati livelli di fattori di crescita, come il VEGF, che stimolano la proliferazione e la migrazione delle cellule endoteliali, favorendo la formazione di nuovi vasi sanguigni [5] extbf(Carmeliet, 2005). Inoltre, la secrezione di MMPs da parte delle cellule tumorali e delle cellule del microambiente tumorale contribuisce alla degradazione della matrice extracellulare, facilitando l'espansione della rete vascolare [25] *(Jain, 2005)*.

I nuovi vasi sanguigni formati all'interno del cancro sono spesso strutturalmente e funzionalmente anomali. Essi tendono ad avere una distribuzione irregolare, pareti sottili e una maggiore permeabilità, che contribuisce a un microambiente tumorale favorevole alla crescita e alla diffusione delle cellule tumorali [46] *(Ribatti, 2010)*. Questa disorganizzazione della rete vascolare rende anche difficile l'accesso dei farmaci antitumorali alle cellule neoplastiche, limitando l'efficacia delle terapie.

Il targeting dell'angiogenesi è diventato una strategia fondamentale nella terapia oncologica. Farmaci anti-angiogenici, come il bevacizumab, un anticorpo monoclonale diretto contro il VEGF, sono stati sviluppati per inibire la formazione di nuovi vasi nei tumori e quindi limitarne la crescita e la metastatizzazione [16] *(Ferrara & Kerbel, 2005)*. Sebbene queste terapie abbiano mostrato risultati promettenti, la resistenza ai farmaci anti-angiogenici rappresenta ancora una sfida significativa e attualmente sono in corso ricerche per sviluppare approcci combinati che possano superare questa resistenza [3] *(Bergers & Hanahan, 2003)*.

1.1.6 Rilevanza dell'angiogenesi nella simulazione e modellizzazione

L'angiogenesi è un processo complesso e dinamico che ha attirato molta attenzione nel campo della simulazione e della modellizzazione computazionale. La modellizzazione dell'angiogenesi permette di comprendere meglio le dinamiche cellulari e molecolari che regolano la formazione di nuovi vasi sanguigni, fornendo strumenti per esplorare come diversi fattori influenzino il processo in contesti fisiologici e patologici [43] *(Peirce, 2008)*.

I modelli computazionali dell'angiogenesi possono essere suddivisi in due principali categorie: modelli di tipo continuo e modelli di tipo discreto. I modelli continui

descrivono l'angiogenesi attraverso equazioni differenziali parziali che rappresentano la densità delle cellule e le concentrazioni dei fattori di crescita. Questi modelli sono particolarmente utili per studiare l'interazione tra il microambiente tumorale e i vasi sanguigni circostanti, nonché per esplorare l'efficacia delle terapie anti-angiogeniche [6] *(Chaplain & Anderson, 1996)*.

D'altra parte, i modelli discreti si concentrano sulla rappresentazione delle singole cellule e delle loro interazioni con l'ambiente circostante. Ad esempio, il *Modello cellulare di Potts* (CPM) è ampiamente utilizzato per simulare la migrazione e la proliferazione delle cellule endoteliali durante l'angiogenesi [31] *(Merks et al., 2006)*. Questo approccio consente di rappresentare dettagliatamente il comportamento cellulare e di investigare come fattori come l'ipossia, i gradienti di fattori di crescita e la rigidità della matrice extracellulare influenzino la morfogenesi dei vasi.

La modellizzazione dell'angiogenesi ha anche un'importanza fondamentale nello sviluppo e nella valutazione delle terapie. I modelli computazionali possono essere utilizzati per testare virtualmente nuove terapie anti-angiogeniche, riducendo il bisogno di sperimentazione in vivo e accelerando il processo di sviluppo dei farmaci [30] *(McDougall et al., 2006)*. Inoltre, questi modelli possono fornire informazioni cruciali su come adattare e ottimizzare le dosi e le combinazioni di trattamenti per ottenere i migliori risultati clinici.

Le simulazioni dell'angiogenesi sono quindi strumenti fondamentali per comprendere meglio sia i meccanismi fisiologici che patologici del processo e per sviluppare nuove strategie terapeutiche. La capacità di modellizzare e simulare l'angiogenesi rappresenta un ponte cruciale tra la ricerca teorica e l'applicazione clinica, contribuendo a migliorare l'efficacia delle terapie per malattie come il cancro e la retinopatia diabetica.

1.2 Modello Cellulare di Potts (CPM)

1.2.1 Introduzione al Modello Cellulare di Potts (CPM)

Il Modello Cellulare di Potts (CPM), introdotto originariamente da Graner e Glazier ([20] *Graner, Glazier, 1992*), è un modello matematico che permette di simulare il comportamento delle cellule biologiche in un contesto di reticolo bidimensionale o tridimensionale. Questo modello è stato pensato per descrivere il comportamento cellulare considerando aspetti come l'adesione cellulare, la deformabilità delle cellule, e l'interazione tra cellule e ambiente circostante. Il CPM è particolarmente utile per analizzare fenomeni come la migrazione cellulare, la morfogenesi dei tessuti e la formazione di pattern, fornendo una rappresentazione dettagliata delle interazioni cellulari basate sull'energia del sistema.

Nel panorama dei modelli matematici per la simulazione dei processi biologici, il CPM si distingue per la sua capacità di rappresentare l'interazione tra cellule in modo probabilistico. Rispetto ad altri modelli, come i modelli *agent-based* (ABM) e i modelli *reazione-diffusione* (RDM), il CPM offre una visione centrata sulla minimizzazione dell'energia del sistema, dove ogni cellula è rappresentata come un insieme di *pixel* (o *voxel* in tre dimensioni) su un reticolo spaziale. Questo approccio permette di catturare la complessità delle interazioni cellulari in termini di adesione e deformazione, descrivendo accuratamente l'evoluzione dinamica delle strutture cellulari [18] (*Glazier, Graner, 1993*).

Storicamente, il CPM è stato utilizzato con successo per simulare la morfogenesi e lo sviluppo embrionale, applicazioni che richiedono una rappresentazione precisa delle forze di adesione tra cellule e della loro dinamica ([2] *Anderson, Quaranta, 2008*). Tuttavia, le prime implementazioni del modello erano limitate in termini di scalabilità e prestazioni, rendendo difficile l'applicazione del CPM per simulazioni su larga scala. Per affrontare queste limitazioni, diversi autori hanno proposto implementazioni parallele del CPM, sfruttando architetture GPGPU (*General-Purpose Graphics Processing Units*) per accelerare le simulazioni [7] (*Chen et al., 2007*).

In questo contesto, il CPM non solo permette di comprendere meglio i meccanismi sottostanti a fenomeni complessi, ma fornisce anche un potente strumento per la modellizzazione e la previsione di comportamenti cellulari in contesti biologici realistici. Grazie alla sua flessibilità, è possibile estendere il modello per includere ulteriori fattori biologici come l'eterogeneità spaziale e la *chemiotassi*, rendendo il CPM uno strumento cruciale per lo studio di sistemi biologici complessi [49] (*Tosin et al., 2006*).

Descrizione del modello CPM

Il Modello Cellulare di Potts (CPM) rappresenta un sistema cellulare come un insieme di pixel disposti su un reticolo spaziale Ω. Ogni punto del reticolo viene associato a un indice che rappresenta la cellula a cui appartiene, permettendo così di identificare le diverse cellule e l'ambiente extracellulare. In un contesto bidimensionale, il reticolo Ω può essere definito come un insieme di punti $\{(i,j) \mid i,j = 1,2,\ldots,N\}$, dove N è la dimensione del reticolo.

A ogni passo temporale, il modello evolve aggiornando la configurazione delle cellule un pixel alla volta, seguendo un insieme di regole probabilistiche che minimizzano l'energia del sistema. Questo processo è gestito tramite l'*algoritmo di Metropolis*, che decide se accettare o meno il cambiamento di stato di un pixel in base alla variazione di energia associata (ΔH). La probabilità di accettazione è definita come:

$$p(\Delta H) = \begin{cases} 1 & \text{se } \Delta H \leq 0 \\ \exp\left(-\frac{\Delta H}{T}\right) & \text{se } \Delta H > 0 \end{cases}$$

Dove T rappresenta un parametro che controlla la probabilità di accettare stati con energia più alta, simile al concetto di temperatura nei sistemi fisici.

Il CPM tiene in considerazione diversi fattori energetici che influenzano il comportamento delle cellule, descritti attraverso la Hamiltoniana del sistema.

La funzione Hamiltoniana è una funzione che rappresenta l'energia totale di un sistema fisico, includendo tutte le forme di energia (come adesione, volume, superficie, ecc.), e viene utilizzata per descrivere l'evoluzione dinamica del sistema, determinando le probabilità degli stati possibili, specialmente nei contesti statistici e nelle simulazioni di Monte Carlo.

Questa è definita come:

$$H = H_{adhesion} + H_{volume} + H_{surface}$$

In particolare:

- $H_{adhesion}$ rappresenta l'energia di *adesione* tra cellule diverse, modellata in base ai contatti tra i pixel di cellule adiacenti.

- H_{volume} è un termine che tiene conto delle differenze tra il *volume* attuale di una cellula e il suo volume target, penalizzando variazioni significative.

- $H_{surface}$ è un termine che considera la superficie cellulare, importante per descrivere la *deformabilità* delle cellule e mantenere la loro forma.

Ogni cellula è rappresentata come un insieme di pixel che condividono lo stesso indice cellulare. La configurazione del sistema evolve nel tempo aggiornando la posizione di singoli pixel, determinando così l'espansione, la contrazione o il movimento delle cellule all'interno del reticolo. Il CPM si distingue per la sua capacità di simulare processi complessi come la morfogenesi, la migrazione cellulare e la chemiotassi, offrendo una rappresentazione dettagliata delle interazioni fisiche e chimiche tra cellule e ambiente.

Oltre alle energie sopracitate, è possibile estendere la funzione Hamiltoniana includendo ulteriori termini energetici, come l'energia di chemiotassi e l'energia elettrica.

Il termine di chemiotassi permette di modellare il movimento diretto delle cellule in risposta a gradienti chimici. Questo viene fatto aggiungendo un termine $H_{chemotaxis}$ all'Hamiltoniana, il quale descrive l'interazione tra le cellule e le concentrazioni chimiche presenti nel reticolo. Questo termine permette di simulare fenomeni biologici complessi, come lo sviluppo embrionale, le risposte immunitarie e la metastasi tumorale, dove il movimento delle cellule è guidato da segnali chimici presenti nell'ambiente.

L'integrazione del termine di energia elettrica consente di modellare l'influenza di un campo elettrico esterno sul comportamento cellulare, aggiungendo un ulteriore livello di complessità al modello CPM. In molti contesti biologici, come la rigenerazione tissutale, la guarigione delle ferite e lo sviluppo embrionale, i campi elettrici giocano un ruolo cruciale nel dirigere il movimento cellulare e la crescita. Includere l'energia elettrica nella Hamiltoniana tramite un termine $H_{electric}$ permette di simulare queste interazioni elettrofisiologiche, migliorando la comprensione dei fenomeni che regolano la dinamica cellulare sotto l'influenza di stimoli elettrici e offrendo una rappresentazione più realistica dei processi biologici complessi [13] (*De Luca, Marcellino, 2024*).

Algoritmo di evoluzione del CPM

L'evoluzione del Modello Cellulare di Potts (CPM) è basata su un processo stocastico che utilizza l'algoritmo di Metropolis per minimizzare l'energia del sistema. L'algoritmo segue una serie di passi per aggiornare la configurazione del reticolo e simulare il comportamento delle cellule nel tempo. Di seguito viene presentato il procedimento dettagliato:

Inizializzazione: Al tempo $\tau = 0$, il reticolo Ω viene inizializzato assegnando a ciascun pixel un indice che indica se il pixel appartiene a una cellula o all'ambiente extracellulare. In questa fase viene anche assegnato un volume target a ciascuna cellula, che servirà per valutare le variazioni di energia.

Selezione del pixel: A ogni passo temporale, viene selezionato casualmente un pixel P all'interno del reticolo e uno dei suoi pixel vicini Q. Questo è tipicamente effettuato utilizzando la *neighborhood di Moore*, che include i pixel adiacenti sia in direzione orizzontale che diagonale.

Calcolo della variazione di energia: Viene calcolata la variazione di energia ΔH che deriverebbe dall'assegnazione dell'indice del pixel vicino Q al pixel selezionato P. La variazione di energia ΔH può essere scritta come:

$$\Delta H = \Delta H_{adhesion} + \Delta H_{volume} + \Delta H_{surface} + \Delta H_{chemotaxis} + \Delta H_{electric}$$

Dove ciascun termine rappresenta la variazione di uno specifico contributo all'Hamiltoniana complessiva del sistema.

Criterio di Metropolis: Una volta calcolata la variazione di energia ΔH, l'algoritmo decide se accettare o meno la modifica proposta in base al criterio di Metropolis. Se $\Delta H \leq 0$, la modifica viene accettata automaticamente in quanto riduce l'energia del sistema. Se $\Delta H > 0$, la modifica viene accettata con probabilità:

$$p(\Delta H) = \exp\left(-\frac{\Delta H}{T}\right)$$

Conosciuta anche come *probabilità di Boltzmann*, dove T è il parametro di temperatura che controlla la probabilità di accettare variazioni energetiche sfavorevoli, introducendo un elemento stocastico nell'evoluzione del sistema.

Aggiornamento dello stato: Se la variazione di energia viene accettata, il pixel P assume l'indice del pixel Q, rappresentando così un'espansione della cellula corrispondente. Questo aggiornamento consente di simulare il movimento cellulare, la crescita e l'interazione tra le cellule.

Iterazione: Il processo viene ripetuto per un numero definito di passi di *Monte Carlo* (*MCS*), dove ciascun passo corrisponde a un numero specifico di tentativi di aggiornamento pari al numero totale di pixel del reticolo. Questo garantisce che ogni pixel abbia una probabilità uniforme di essere selezionato a ogni passo temporale.

L'algoritmo di evoluzione del CPM è fondamentale per simulare fenomeni biologici complessi, come la formazione di strutture tissutali, la migrazione cellulare e la risposta a segnali chimici attraverso la chemiotassi. Grazie all'integrazione del termine di chemiotassi nell'Hamiltoniana, il CPM può modellare in modo realistico il movimento delle cellule verso aree con concentrazioni più elevate di segnali chimici [13] (*De Luca, Marcellino, 2024*).

Energia del sistema e definizione dell'Hamiltoniana

La Hamiltoniana del Modello Cellulare di Potts (CPM) rappresenta l'energia totale del sistema e gioca un ruolo fondamentale nella determinazione del comportamento delle cellule durante l'evoluzione temporale. La minimizzazione di questa energia guida le cellule a evolvere verso configurazioni più stabili, considerando diversi fattori energetici che influenzano le interazioni tra cellule e con l'ambiente. L'Hamiltoniana è definita come la somma di diversi contributi energetici, ciascuno dei quali rappresenta un aspetto specifico del comportamento cellulare:

$$H = H_{adhesion} + H_{volume} + H_{surface} + H_{chemotaxis} + H_{electric}$$

Dove:

1. **Energia di adesione ($H_{adhesion}$)**:

   Questo termine rappresenta l'energia associata ai contatti tra cellule diverse e tra cellule e ambiente extracellulare. Viene modellato utilizzando un parametro di

adesione J, che controlla la forza dell'interazione tra diverse cellule o tra cellule e il mezzo.

$$H_{adhesion} = \sum_{\mathbf{x}} \sum_{\mathbf{x}' \in N(\mathbf{x})} J(\tau(\sigma(\mathbf{x})), \tau(\sigma(\mathbf{x}')))(1 - \delta(\sigma(\mathbf{x}), \sigma(\mathbf{x}')))$$

Dove $N(\mathbf{x})$ rappresenta l'insieme dei vicini del pixel $\mathbf{x}$, $J(\tau(\sigma(\mathbf{x})), \tau(\sigma(\mathbf{x}')))$ è l'energia di adesione tra i tipi cellulari $\tau(\sigma(\mathbf{x}))$ e $\tau(\sigma(\mathbf{x}'))$, e $\delta(\sigma(\mathbf{x}), \sigma(\mathbf{x}'))$ è il *delta di Kronecker* che vale 1 se i due tipi cellulari sono uguali e 0 altrimenti. Questo termine modella la tendenza delle cellule a minimizzare l'energia superficiale, favorendo l'adesione tra cellule simili [18] *(Glazier, Graner, 1993)*.

2. **Energia di volume (H_{volume})**:

   Le cellule tendono a mantenere un volume costante durante il loro comportamento biologico. Questo è modellato nel CPM utilizzando un termine di penalizzazione per le deviazioni dal volume target.

   $$H_{volume} = \lambda_V \sum_{\sigma > 0} (V(\sigma) - V_{target}(\sigma))^2$$

   Dove $V(\sigma)$ è il volume attuale della cellula σ, $V_{target}(\sigma)$ è il volume target della cellula, e λ_V è un moltiplicatore di Lagrange che controlla la forza del vincolo di volume. Questo termine assicura che le cellule mantengano dimensioni realistiche durante la simulazione [18] *(Glazier, Graner, 1993)*.

3. **Energia di superficie ($H_{surface}$)**:

   Questo termine è responsabile del controllo della superficie della cellula, penalizzando le deviazioni rispetto alla superficie target. Tale parametro è importante per descrivere la deformabilità delle cellule e assicurare che la loro forma rimanga coerente con le caratteristiche biologiche.

   $$H_{surface} = \lambda_S \sum_{\sigma > 0} (S(\sigma) - S_{target}(\sigma))^2$$

   Dove $S(\sigma)$ è la superficie attuale della cellula σ, $S_{target}(\sigma)$ è la superficie target, e λ_S è un moltiplicatore di Lagrange che determina l'importanza del vincolo sulla superficie.

4. **Energia di chemiotassi ($H_{chemotaxis}$)**:

   La chemiotassi rappresenta la capacità delle cellule di muoversi in risposta a gradienti chimici. Questo termine viene aggiunto all'Hamiltoniana per modellare il comportamento delle cellule che tendono a spostarsi verso aree con concentrazioni più elevate di determinati segnali chimici.

   $$H_{chemotaxis} = -\lambda_C \sum_x \sum_i \mu_i(\tau(\sigma(x)))c_i(x)$$

   Dove λ_C è un parametro globale che controlla la forza della chemiotassi, $\mu_i(\tau)$ rappresenta la sensibilità chemiotattica del tipo di cellula τ al composto chimico

i, e $c_i(x)$ è la concentrazione del composto chimico i alla posizione x. Questo termine permette di simulare fenomeni biologici come la migrazione cellulare guidata da segnali chimici [13] *(De Luca, Marcellino, 2024)*.

5. **Energia elettrica ($H_{electric}$)**:

   Questo termine rappresenta l'effetto di un campo elettrico esterno sul comportamento delle cellule. Viene aggiunto alL'Hamiltoniana per modellare l'interazione delle cellule con un campo elettrico applicato, influenzando la loro distribuzione e movimento.

$$H_{electric} = -\lambda_E \sum_{\mathbf{x}} \mathbf{E}(\mathbf{x}) \cdot \mathbf{x}$$

   Dove λ_E è un parametro globale che rappresenta l'intensità del campo elettrico, $E(x)$ è il vettore del campo elettrico nella posizione x e x è il vettore posizione.

Questi termini energetici contribuiscono alla definizione dello stato del sistema e determinano il comportamento delle cellule durante l'evoluzione temporale. La minimizzazione dell'Hamiltoniana, attraverso l'algoritmo di Metropolis, consente di simulare in maniera realistica processi biologici come l'adesione cellulare, la morfogenesi e la risposta a segnali chimici [18] *(Glazier, Graner, 1993)*.

1.2.2 Estensione del CPM per includere la chemiotassi e il campo elettrico

La chemiotassi rappresenta un meccanismo fondamentale per la migrazione cellulare, in cui le cellule si muovono in risposta a gradienti di concentrazione di sostanze chimiche. Questa estensione è essenziale per rappresentare accuratamente processi biologici come lo sviluppo embrionale, le risposte immunitarie e la metastasi tumorale. Per incorporare la chemiotassi nel Modello Cellulare di Potts (CPM), è necessario introdurre un termine aggiuntivo nella funzione Hamiltoniana che descriva l'interazione tra le cellule e i gradienti chimici presenti nell'ambiente circostante.

Un'ulteriore estensione del CPM consiste nell'integrazione dell'energia associata a un campo elettrico, per modellare l'interazione delle cellule con un campo elettrico esterno. L'energia del campo elettrico, similmente alla chemiotassi, aggiunge un livello di complessità che permette di simulare fenomeni come l'elettrotassi, che è la risposta diretta delle cellule a stimoli elettrici. Questo è fondamentale per descrivere processi biologici quali la rigenerazione dei tessuti, la guarigione delle ferite e lo sviluppo embrionale, dove i campi elettrici influenzano il movimento e la direzione delle cellule.

Definizione del termine di chemiotassi

Per modellare la chemiotassi nel CPM, viene aggiunto un termine $H_{chemotaxis}$ alla funzione Hamiltoniana del sistema, che descrive la tendenza delle cellule a muoversi verso regioni con concentrazioni più elevate di specifici segnali chimici [13] (*De Luca, Marcellino, 2024*).

$$H_{chemotaxis} = -\lambda_C \sum_x \sum_i \mu_i(\tau(\sigma(x)))c_i(x)$$

Dove λ_C è un parametro globale che controlla la forza della chemiotassi, $\mu_i(\tau)$ rappresenta la sensibilità chemiotattica del tipo di cellula τ al composto chimico i, e $c_i(x)$ è la concentrazione del composto chimico i alla posizione x. Il termine $H_{chemotaxis}$ ha un segno negativo per garantire che le cellule si muovano verso regioni con alte concentrazioni di segnali chemotattici (quando $\mu_i > 0$), mentre si allontanano dai gradienti di repellenti (quando $\mu_i < 0$). Questo approccio permette di modellare sia *attrattanti* che *repellenti* all'interno del sistema, offrendo una rappresentazione più realistica del comportamento cellulare.

Implementazione del gradiente chimico

Per implementare la chemiotassi nel CPM, è necessario calcolare i gradienti chimici nel reticolo. Una tecnica comunemente utilizzata è l'approssimazione mediante *differenze finite*. Il gradiente di concentrazione $\nabla c_i(x)$ può essere approssimato come:

$$\nabla c_i(x) \approx \frac{1}{2a} \sum_{x' \in N(x)} (x' - x)(c_i(x') - c_i(x))$$

Dove a rappresenta la distanza tra i punti del reticolo, e $N(x)$ denota i vicini del punto x. Questa approssimazione permette di calcolare l'influenza del gradiente chimico sulla dinamica cellulare, rendendo possibile l'integrazione del termine di chemiotassi all'interno dell'algoritmo di Metropolis.

Evoluzione delle concentrazioni chimiche

Per completare il modello, è necessario descrivere l'evoluzione delle concentrazioni chimiche nel tempo. Questo viene fatto utilizzando un'equazione di reazione-diffusione che rappresenta la dinamica delle sostanze chimiche nell'ambiente:

$$\frac{\partial c_i}{\partial t} = D_i \nabla^2 c_i + R_i(c_1, \ldots, c_n, \sigma)$$

Dove:

- D_i è il coefficiente di diffusione del composto chimico i.
- R_i rappresenta i termini di reazione che possono dipendere dalle concentrazioni degli altri composti chimici e dalla distribuzione delle cellule.

Questa equazione può essere risolta numericamente utilizzando metodi di differenze finite per calcolare le nuove concentrazioni chimiche ad ogni passo temporale, alternandosi con l'aggiornamento della configurazione cellulare tramite l'algoritmo CPM. In questo modo, è possibile simulare un sistema in cui le cellule rispondono dinamicamente ai cambiamenti dei gradienti chimici.

Definizione del termine di energia del campo elettrico

Analogamente alla chemiotassi, l'integrazione di un campo elettrico nel CPM richiede l'introduzione di un termine aggiuntivo nella funzione Hamiltoniana, indicato con $H_{electric}$. Il termine di energia del campo elettrico è definito come:

$$H_{electric} = -\lambda_E \sum_{\mathbf{x}} \mathbf{E}(\mathbf{x}) \cdot \mathbf{x}$$

Dove λ_E rappresenta un parametro globale che controlla l'intensità del campo elettrico, $\mathbf{E}(\mathbf{x})$ è il vettore del campo elettrico alla posizione $\mathbf{x}$, e $\mathbf{x}$ è il vettore posizione.

Per calcolare il campo elettrico, è necessario risolvere un sistema lineare basato sull'equazione di Poisson:

$$\nabla^2 \phi = -\rho$$

dove ρ rappresenta la densità di carica, che può essere proporzionale alla concentrazione chimica presente nel sistema:

$$\rho(x) = \alpha c(x)$$

Discretizzando l'operatore Laplaciano utilizzando il metodo delle differenze finite, otteniamo:

$$\nabla^2 \phi_{i,j} \approx \frac{\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1} - 4\phi_{i,j}}{(\Delta x)^2}$$

Questo porta a un sistema lineare della forma:

$$A\phi = b$$

dove A è una matrice sparsa che rappresenta la Laplaciana discretizzata, ϕ è il vettore dei valori del potenziale elettrico e b è il vettore delle densità di carica scalate da $-(\Delta x)^2$. Una volta risolto il sistema per ottenere il potenziale elettrico ϕ, il campo elettrico $\mathbf{E}$ viene calcolato come il gradiente negativo del potenziale:

$$\mathbf{E} = -\nabla \phi$$

Sul reticolo, il gradiente viene approssimato utilizzando le *differenze centrali*:

$$E_x(i,j) \approx -\frac{\phi_{i+1,j} - \phi_{i-1,j}}{2\Delta x}$$

$$E_y(i,j) \approx -\frac{\phi_{i,j+1} - \phi_{i,j-1}}{2\Delta x}$$

Integrazione con la dinamica del CPM

Il termine di energia del campo elettrico viene poi integrato nella Hamiltoniana del CPM per descrivere l'influenza del campo elettrico esterno sul movimento cellulare. L'energia associata al campo elettrico contribuisce alla funzione totale dell'Hamiltoniana e viene utilizzata nell'algoritmo di Metropolis per determinare la probabilità di accettare o rifiutare una proposta di movimento delle cellule.

Quando un sito del reticolo $\mathbf{x}$ viene proposto per cambiare il suo indice verso quello di un vicino $\mathbf{x}'$, la variazione di energia associata al campo elettrico è data da:

$$\Delta H_{electric} = -\lambda_E [\mathbf{E}(\mathbf{x}') \cdot \mathbf{x}' - \mathbf{E}(\mathbf{x}) \cdot \mathbf{x}]$$

Questa variazione di energia viene incorporata nell'algoritmo di Metropolis insieme agli altri termini energetici, influenzando la probabilità di accettare il movimento proposto. In questo modo, il campo elettrico agisce direttamente sulla dinamica del sistema, guidando il movimento delle cellule secondo il gradiente di potenziale elettrico e favorendo la comprensione dei fenomeni elettrofisiologici che influenzano la loro dinamica.

Importanza dell'estensione del campo elettrico

L'integrazione dell'energia del campo elettrico nel CPM consente di simulare fenomeni biologici complessi in cui le cellule rispondono a stimoli elettrofisiologici. Questo è particolarmente rilevante in contesti come la rigenerazione dei tessuti e la guarigione delle ferite, dove i campi elettrici sono noti per giocare un ruolo cruciale nel dirigere il movimento delle cellule. Aggiungendo il termine $H_{electric}$ alla funzione Hamiltoniana, il modello CPM esteso può rappresentare con maggiore accuratezza le interazioni tra cellule e campi elettrici, offrendo una visione più dettagliata dei meccanismi che regolano la dinamica cellulare sotto l'influenza di stimoli elettrici.

Profilo energetico

L'aggiunta del termine di chemiotassi modifica significativamente il profilo energetico del CPM, influenzando la probabilità di accettare o rifiutare una proposta di movimento delle cellule. La variazione di energia associata a una proposta di aggiornamento del sito $\mathbf{x}$ verso il vicino $\mathbf{x}'$ è data da:

$$\Delta H_{chemotaxis} = -\lambda_C \sum_i [\mu_i(\tau(\sigma(\mathbf{x}'))) - \mu_i(\tau(\sigma(\mathbf{x})))]c_i(\mathbf{x})$$

Questa espressione mostra che il cambiamento di energia dipende dalla concentrazione chimica locale $c_i(\mathbf{x})$ e dalla differenza nella sensibilità chemotattica μ_i tra il tipo cellulare del sito originale e quello del sito proposto.

In particolare:

- La variazione energetica è proporzionale alla concentrazione chimica locale, il che implica che le cellule tenderanno a muoversi verso aree con gradiente positivo di attrattanti.

- La magnitudo del cambiamento di energia dipende dalla differenza nella sensibilità chemotattica tra le cellule coinvolte nello scambio. Questo permette di modellare comportamenti complessi, come la competizione tra diversi tipi cellulari per i gradienti chimici.

- Se tutti i tipi cellulari hanno la stessa sensibilità chemotattica, il termine di chemiotassi influenzerà solo il movimento collettivo della popolazione cellulare, senza causare interazioni preferenziali tra cellule di diversi tipi.

L'analisi del profilo energetico aiuta a comprendere come le cellule rispondano ai gradienti chimici e come queste risposte siano influenzate dalle proprietà chimiche e dalle caratteristiche intrinseche delle cellule, offrendo una visione più chiara delle dinamiche emergenti dalla simulazione.

Connessione con il Modello di Keller-Segel

L'estensione del CPM per includere la chemiotassi può essere collegata al *modello di Keller-Segel* ([27] *Keller, Segel, 1971*), un modello classico di chemiotassi che descrive il comportamento collettivo delle cellule in risposta a segnali chimici. Nel limite continuo, il termine di chemiotassi del CPM può essere scritto come:

$$H_{chemotaxis} = -\lambda_C \int_\Omega \sum_\tau \sum_i \mu_i(\tau) c_i(x) \rho_\tau(x) dx$$

Dove $\rho_\tau(x)$ rappresenta la densità locale delle cellule di tipo τ. In questo modo, il CPM esteso per la chemiotassi può essere visto come una versione discretizzata del modello di Keller-Segel, fornendo una connessione teorica tra il comportamento cellulare simulato e le descrizioni matematiche classiche della chemiotassi.

Applicazioni biologiche

L'integrazione della chemiotassi nel CPM è cruciale per rappresentare accuratamente vari processi biologici. Ad esempio:

- **Sviluppo embrionale**: le cellule si muovono verso segnali che determinano la formazione delle strutture tissutali.
- **Risposte immunitarie**: i leucociti migrano verso i siti di infezione seguendo gradienti di chemochine.
- **Metastasi tumorale**: le cellule cancerogene si spostano lungo gradienti chimici per invadere nuovi tessuti.

Questi fenomeni possono essere simulati in modo dettagliato grazie all'estensione chemiotattica del CPM, fornendo una comprensione più approfondita dei meccanismi sottostanti alla migrazione cellulare e alle interazioni con l'ambiente circostante.

1.2.3 Ruolo dei Metodi agli Elementi Finiti (FEM) nella CPM

I *Metodi agli Elementi Finiti* (FEM) sono strumenti numerici molto potenti che possono essere utilizzati per migliorare l'accuratezza delle simulazioni nel Modello Cellulare di Potts (CPM), in particolare per descrivere meglio le interazioni meccaniche all'interno e tra le cellule. Nel contesto del CPM, i metodi agli elementi finiti sono principalmente impiegati per calcolare le deformazioni e le tensioni cellulari, fornendo una rappresentazione più dettagliata delle dinamiche meccaniche coinvolte nei processi biologici ([51] *Zienkiewicz, Taylor, 2000*.

Introduzione ai Metodi agli Elementi Finiti

I Metodi agli Elementi Finiti (FEM) sono utilizzati per risolvere equazioni differenziali parziali (PDE) in geometrie complesse suddividendo il dominio in sottoelementi più piccoli, detti elementi finiti. Ciascun elemento è descritto da una funzione di forma che permette di approssimare le soluzioni delle PDE. Questa metodologia è particolarmente utile per modellare la meccanica tissutale e altre caratteristiche meccaniche che non possono essere facilmente rappresentate con la sola configurazione del reticolo del CPM.

Applicazione dei FEM nel CPM

Nel CPM, i metodi agli elementi finiti vengono applicati per migliorare la rappresentazione delle interazioni meccaniche tra le cellule e l'ambiente extracellulare. In particolare, i FEM sono utilizzati per [44] (*Rejniak, 2007*):

- **Calcolare le deformazioni cellulari**: Le cellule sono soggette a deformazioni derivanti dalle interazioni con altre cellule e con l'ambiente. Utilizzando i FEM, è possibile modellare come le cellule si deformano e reagiscono alle forze esterne, calcolando accuratamente la distribuzione della tensione all'interno delle cellule.

- **Assemblare la matrice di rigidezza**: La matrice di rigidezza viene utilizzata per descrivere la resistenza delle cellule alla deformazione. Questa matrice è costruita utilizzando le proprietà del materiale cellulare e la geometria delle cellule, ed è un componente fondamentale per descrivere le risposte meccaniche delle cellule alle forze.

- **Calcolare le forze di adesione**: I FEM possono essere utilizzati per determinare le forze di adesione tra cellule, fornendo una rappresentazione più accurata delle interazioni cellulari rispetto alla semplice somma delle energie di adesione tra i pixel del CPM.

Considerazioni numeriche

L'integrazione dei FEM nel CPM richiede alcune considerazioni numeriche per garantire la stabilità e l'accuratezza della simulazione:

- **Discretizzazione del dominio**: Il dominio di ciascuna cellula deve essere discretizzato in elementi finiti. La scelta della dimensione degli elementi influisce direttamente sull'accuratezza della simulazione e sul tempo computazionale richiesto.

- **Condizioni sulla frontiera**: Le condizioni sulla frontiera devono essere specificate per ciascun elemento per risolvere le equazioni differenziali che descrivono le deformazioni cellulari. Queste condizioni possono includere vincoli di adesione con altre cellule o con l'ambiente extracellulare.

- **Efficienza computazionale**: L'uso dei FEM nel CPM aumenta la complessità computazionale del modello. Per questo motivo, spesso si utilizzano tecniche di parallelizzazione, come l'uso delle GPU, per accelerare i calcoli e rendere possibile la simulazione di sistemi biologici su larga scala.

Applicazioni biologiche dei FEM nel CPM

L'integrazione dei metodi agli elementi finiti nel CPM è particolarmente utile per rappresentare processi biologici che coinvolgono deformazioni meccaniche significative. Alcune applicazioni includono:

- **Morfogenesi tissutale**: La morfogenesi è un processo durante il quale i tessuti si formano e si organizzano in strutture complesse. L'uso dei FEM permette di modellare le forze meccaniche coinvolte nella formazione e nella differenziazione dei tessuti.

- **Migrazione cellulare**: Durante la migrazione, le cellule possono deformarsi per adattarsi alle superfici su cui si muovono o per superare barriere fisiche. I FEM consentono di modellare accuratamente queste deformazioni e di prevedere come le cellule reagiranno a vari stimoli meccanici.

- **Interazioni con la matrice extracellulare (ECM)**: Le cellule interagiscono con l'ECM attraverso forze meccaniche. I FEM permettono di rappresentare come queste interazioni influenzano la dinamica cellulare, modellando sia la deformazione delle cellule che la risposta dell'ECM.

L'uso dei Metodi agli Elementi Finiti nel CPM amplia le possibilità di modellazione dei fenomeni biologici, consentendo di includere effetti meccanici complessi che altrimenti verrebbero trascurati. Questa integrazione fornisce una visione più completa del comportamento cellulare, combinando le interazioni meccaniche e chimiche per simulare processi biologici su larga scala.

1.2.4 Implementazione del CPM in CUDA

La crescente necessità di simulare fenomeni biologici su larga scala ha reso indispensabile l'uso di tecniche di parallelizzazione per accelerare il Modello Cellulare di Potts (CPM). La *Computing Unified Device Architecture* (CUDA) di NVIDIA offre una piattaforma efficiente per la parallelizzazione del CPM, sfruttando la potenza delle GPU (*Graphics Processing Units*) per eseguire simulazioni complesse con un numero elevato di cellule e iterazioni.

Motivazione per l'uso delle GPU

Il CPM è un modello computazionalmente intensivo, poiché ogni aggiornamento del sistema richiede il calcolo della variazione di energia per ogni pixel e ogni interazione tra i pixel adiacenti. Questo porta a una complessità computazionale elevata, specialmente in sistemi di grandi dimensioni. L'uso delle GPU permette di:

- **Parallelizzare i calcoli**: Le GPU sono progettate per eseguire migliaia di thread in parallelo, permettendo di calcolare simultaneamente la variazione di energia per diversi pixel del reticolo.
- **Ridurre i tempi di simulazione**: La parallelizzazione consente di ridurre significativamente i tempi di simulazione rispetto a un'implementazione sequenziale su CPU.
- **Scalabilità**: Le GPU possono gestire un numero elevato di elementi in parallelo, rendendo possibile la simulazione di sistemi biologici su larga scala.

Struttura dell'implementazione in CUDA

L'implementazione del CPM in CUDA richiede la suddivisione delle operazioni in diversi *kernel* che vengono eseguiti in parallelo sulla GPU. Di seguito sono descritti i principali passi per l'implementazione del CPM in CUDA:

- **Inizializzazione del reticolo**

  Il reticolo Ω è rappresentato come una matrice bidimensionale di interi, dove ogni elemento indica l'indice della cellula corrispondente. Questa matrice è allocata sia sulla CPU (*host*) che sulla GPU (*device*), e viene inizializzata per rappresentare la configurazione iniziale del sistema.

- **Calcolo della variazione di energia**

  Il calcolo della variazione di energia ΔH per ogni pixel nella versione corrente non presenta una parallelizzazione, poiché questo viene determinato nella parte dell'algoritmo di Metropolis, la cui componente stocastica genera un meccanismo di interdipendenza tra le cellule, dovute alla possibilità di accettazione dello stato di transizione. Questo perché, qualora lo stato fosse accettato, le cellule vicine ne sarebbero necessariamente influenzate per il loro calcolo di energia.

- **Criterio di Metropolis**

  Una volta calcolata la variazione di energia, viene applicato il criterio di Metropolis per determinare se accettare o meno il cambiamento di stato del pixel. Questo criterio viene implementato utilizzando una funzione randomica che è generata per ogni thread, garantendo così un'evoluzione stocastica del sistema.

- **Aggiornamento del reticolo**

  I pixel che soddisfano il criterio di Metropolis vengono aggiornati simultaneamente. Questo è un passaggio critico che richiede l'uso di tecniche di sincronizzazione per evitare conflitti tra thread che potrebbero aggiornare lo stesso pixel contemporaneamente.

- **Gestione della concorrenza**

  Per garantire la correttezza della simulazione, è necessario gestire la concorrenza tra *thread*. L'implementazione CUDA utilizza blocchi di thread per suddividere il reticolo in sottoinsiemi, in modo che i conflitti vengano minimizzati e le operazioni siano sincronizzate correttamente.

Allocazione della memoria e trasferimento dei dati

L'allocazione della memoria è un aspetto cruciale nell'implementazione del CPM in CUDA. I dati devono essere trasferiti tra la memoria dell'host (CPU) e la memoria del device (GPU) in modo efficiente per ridurre i tempi di latenza. Le seguenti operazioni sono essenziali:

- **Allocazione della memoria sulla GPU**: Le matrici che rappresentano il reticolo, le energie e le variabili ausiliarie devono essere allocate sulla GPU utilizzando la funzione `cudaMalloc()`.

- **Copia dei dati tra Hhst e device**: I dati iniziali vengono copiati dall'host al device utilizzando `cudaMemcpy()`. Durante la simulazione, è necessario minimizzare la frequenza di trasferimento dei dati tra host e device, poiché questo processo è relativamente lento rispetto ai calcoli sulla GPU.

- **Deallocazione della memoria**: Alla fine della simulazione, la memoria allocata sulla GPU deve essere liberata con 'cudaFree()' per evitare *memory leaks*.

Ottimizzazioni dell'implementazione CUDA

Per massimizzare le prestazioni dell'implementazione CUDA del CPM, sono possibili diverse ottimizzazioni:

- **Coalescenza dell'accesso alla memoria**: Per migliorare le prestazioni della GPU, è importante che i thread accedano alla memoria globale in modo coalescente, ossia che accedano a indirizzi di memoria contigui. Questo permette di ridurre i tempi di accesso alla memoria.

- **Uso della memoria condivisa**: La memoria condivisa è una memoria veloce che può essere utilizzata dai thread all'interno dello stesso blocco. Utilizzare la memoria condivisa per memorizzare i valori dei pixel vicini riduce il numero di accessi alla memoria globale, migliorando le prestazioni.

- **Minimizzazione della divergenza dei thread**: La divergenza dei thread si verifica quando i thread all'interno dello stesso *warp* seguono percorsi di esecuzione diversi. Questo problema è minimizzato strutturando il codice in modo che i rami condizionali siano coerenti tra i thread di un warp.

Sfide nell'implementazione del CPM in CUDA

Nonostante i vantaggi offerti dalla parallelizzazione GPU, ci sono diverse sfide nella sua implementazione con CUDA:

- **Conflitti di aggiornamento**: La natura parallela del CPM su GPU introduce la possibilità di conflitti durante l'aggiornamento simultaneo dei pixel. Tecniche di gestione della concorrenza come l'uso di funzioni atomiche o la suddivisione del reticolo in zone indipendenti sono necessarie per mitigare questi problemi.

- **Limitazioni di memoria**: La memoria disponibile sulla GPU è limitata rispetto a quella della CPU. Pertanto, per simulazioni molto grandi, potrebbe essere necessario suddividere il problema in sottoproblemi più piccoli o utilizzare tecniche di multi-GPU.

- **Difficoltà di debugging**: Il debugging di codice CUDA può essere più complesso rispetto al codice CPU tradizionale, richiedendo strumenti specializzati come *cuda-gdb* o *compute-sanitizer*.

Per concludere, l'implementazione del CPM in CUDA rappresenta un passo fondamentale per l'analisi di fenomeni biologici complessi su larga scala. Grazie alla potenza di calcolo delle GPU, è possibile eseguire simulazioni dettagliate in tempi ragionevoli, consentendo una maggiore comprensione dei processi biologici che regolano il comportamento cellulare.

Capitolo 2

Metodologia: dal sequenziale al parallelo

L'implementazione di un modello di simulazione inizia spesso con un approccio sequenziale, poiché questo risulta più semplice da sviluppare e analizzare. Tuttavia, un approccio sequenziale diventa presto inefficace quando si affrontano simulazioni su larga scala, in cui la complessità e il carico computazionale crescono rapidamente. In questo capitolo viene descritto il percorso evolutivo del modello di simulazione Monte Carlo, partendo dall'implementazione sequenziale in *MATLAB* e passando attraverso una successiva reimplementazione in *C*, per sfruttare l'efficienza di un linguaggio di programmazione a basso livello e rendere più agevole la futura conversione in *CUDA*, concludendo con un adattamento delle routine di visualizzazione di MATLAB in *C++* facendo uso della libreria *OpenCV*.

Il punto di partenza è stato un modello fornito, interamente implementato in MATLAB, utilizzato per verificare la correttezza dell'algoritmo e comprendere le interazioni tra i vari contributi energetici. Questa versione iniziale è stata cruciale per sviluppare una comprensione profonda della dinamica del sistema e delle esigenze computazionali del problema.

Questo capitolo è dunque strutturato in tre parti principali: la prima descrive il codice in MATLAB, con i suoi limiti; la seconda analizza l'implementazione riscritta in C, evidenziando i vantaggi dell'utilizzo di un linguaggio a basso livello; e la terza parte si concentra sulle routine di visualizzazione, convertite in C++ con l'ausilio della libreria OpenCV. Ogni fase di questo percorso ha contribuito in modo significativo all'evoluzione del modello, garantendo un miglioramento progressivo delle prestazioni e un'ottimizzazione della rappresentazione visiva.

2.1 Prima implementazione in MATLAB

La prima versione del modello di simulazione Monte Carlo è stata implementata in MATLAB. Questo codice rappresentava il punto di partenza per lo sviluppo e la validazione del modello, permettendo di testare l'efficacia delle diverse componenti dell'Hamiltoniana, come l'adesione cellulare, il volume delle cellule, e l'inclusione dei gradienti chimici per la chemiotassi.

Il codice originale fornito non includeva un meccanismo per la lettura di dati generati da altri programmi. Per facilitare una fase di visualizzazione intermedia, è stata aggiunta la possibilità di leggere due file contenenti il reticolo e il campo chimico, ge-

nerati dal codice in C. Questo adattamento è stato fondamentale per poter visualizzare le simulazioni in maniera efficace, specialmente nella fase di verifica dei risultati del porting del codice da MATLAB a C.

Le routine di visualizzazione originali in MATLAB, pur essendo utili per una rappresentazione preliminare dei risultati, hanno mostrato limiti significativi in termini di prestazioni.

La prima implementazione in MATLAB si è quindi rivelata essenziale per costruire una base solida su cui sviluppare ulteriori ottimizzazioni, sia in termini di codice sequenziale in C che di visualizzazione in C++.

2.1.1 Descrizione del file ricevuto e del suo contenuto

Il file `cpm_angio_2.m` rappresenta la prima versione del modello di simulazione Monte Carlo implementato in MATLAB. Il codice si concentra sulla simulazione del comportamento delle cellule in un reticolo bidimensionale, utilizzando il Modello Cellulare di Potts (CPM) per rappresentare le interazioni energetiche tra le cellule e l'ambiente circostante.

Il file si compone di diverse sezioni principali, tra cui:

1. **Inizializzazione del sistema**:
   In questa fase, il codice crea il reticolo, assegna un identificativo alle varie celle e inizializza i campi chimici. Il reticolo è rappresentato come una matrice bidimensionale, e ciascun sito della matrice può appartenere a una cellula specifica o all'ambiente extracellulare.

2. **Simulazione Monte Carlo**:
   La parte centrale del codice implementa un ciclo Monte Carlo, in cui viene ripetuto un certo numero di passi (Monte Carlo Steps, MCS). Ad ogni passo, vengono selezionati in modo casuale un sito del reticolo e un suo vicino, e si valuta il cambiamento di energia che deriverebbe dallo scambio tra i due siti, includendo contributi energetici come l'adesione, il volume, la superficie, la chemiotassi, e l'energia elettrica.

3. **Routine di visualizzazione**:
   La versione originale del codice includeva anche una parte dedicata alla visualizzazione dei risultati, utilizzando le funzionalità grafiche di MATLAB. Per permettere una verifica visiva dei dati generati successivamente dal porting in C, è stata aggiunta la possibilità di leggere da dei file già esistenti (generati dal codice in C) contenenti il reticolo e il campo chimico, al fine di permettere una prima visualizzazione dei dati.

La struttura del codice ricevuto ha quindi permesso di costruire una base su cui operare ottimizzazioni e miglioramenti, in particolare in termini di visualizzazione e gestione delle risorse computazionali, ed in particolare per una successiva versione parallela e per l'uso di librerie più efficienti anche per la visualizzazione, come OpenCV.

2.1.2 Funzionamento del modello sequenziale in MATLAB

Prima di procedere con la spiegazione dell'implementazione, è bene introdurre alcuni concetti, quali l'algoritmo di Metropolis e la Simulazione di Monte Carlo.

Algoritmo di Metropolis

L'algoritmo di Metropolis è un metodo stocastico utilizzato per generare campioni da una distribuzione di probabilità desiderata, ed è particolarmente utile in contesti dove il calcolo diretto della distribuzione risulta complicato o computazionalmente proibitivo [32] (*Nicholas Metropolis et al.*, *1953*). Nel contesto della simulazione di Monte Carlo, l'algoritmo di Metropolis consente di determinare se una certa modifica allo stato del sistema debba essere accettata, in modo da minimizzare l'energia totale del sistema o rispettare la distribuzione di equilibrio termodinamico.

L'algoritmo procede confrontando l'energia attuale del sistema con quella risultante dalla possibile transizione. Se la variazione di energia ΔH è minore o uguale a zero, la transizione viene sempre accettata, in quanto diminuisce l'energia del sistema. Se invece $\Delta H > 0$, la transizione viene accettata con una probabilità $e^{-\Delta H/T}$, dove T rappresenta la temperatura del sistema [29] (*S. Kirkpatrick, C. D. Gelatt e M. P. Vecchi*, *1983*). Questa probabilità introduce la possibilità di accettare transizioni che aumentano l'energia del sistema, in modo da evitare di rimanere bloccati in minimi locali e permettere una migliore esplorazione dello spazio delle configurazioni. L'algoritmo di Metropolis è quindi un elemento fondamentale per garantire che la simulazione possa esplorare in maniera efficace l'intero spazio delle possibili configurazioni del sistema, bilanciando l'ottimizzazione energetica con la diversità delle configurazioni.

Simulazione di Monte Carlo

Una simulazione di Monte Carlo è una tecnica computazionale utilizzata per modellare e analizzare sistemi complessi, la cui evoluzione dipende da processi probabilistici [26] (*M. H. Kalos e P. A. Whitlock*, *2008*). Nel contesto del Modello Cellulare di Potts, la simulazione Monte Carlo è utilizzata per rappresentare la dinamica delle cellule e la loro interazione con l'ambiente circostante [4] (*Kurt Binder e Dieter W. Heermann*, *2010*). In una simulazione di Monte Carlo, ogni passaggio viene effettuato scegliendo casualmente un elemento del sistema (nel caso del CPM, un pixel del reticolo) e applicando una serie di regole probabilistiche per determinare come modificare lo stato del sistema stesso.

La simulazione Monte Carlo permette di esplorare in maniera statistica lo spazio delle configurazioni del sistema, stimando come le cellule evolvono nel tempo. Questo approccio è particolarmente efficace quando si trattano sistemi biologici complessi, dove il comportamento emergente è frutto di numerose interazioni locali tra cellule, il cui esito non può essere determinato in maniera deterministica. La natura stocastica del metodo rende possibile simulare fenomeni come la morfogenesi, la migrazione cellulare e la adesione cellulare, che sono tutti influenzati da fattori casuali.

L'algoritmo di Metropolis è parte integrante della simulazione Monte Carlo, in quanto stabilisce le regole per l'accettazione delle transizioni, consentendo al sistema di convergere verso configurazioni a più bassa energia in maniera stocastica [32] (*Nicholas Metropolis et al.*, *1953*). Questo approccio, che permette al sistema di evitare minimi locali e migliorare il processo di ottimizzazione, rende le simulazioni Monte Carlo ideali per rappresentare la dinamica cellulare in ambienti complessi e per studiare fenomeni emergenti su larga scala.

Pseudocodice sequenziale MATLAB

Il funzionamento del modello è stato schematizzato nello pseudocodice riportato di seguito, che rappresenta le diverse fasi della simulazione:

Algoritmo 1: Modello Cellulare di Potts (MATLAB)

Input:

`LATTICE_SIZE`	Dimensione del reticolo
`NUM_CELLS`	Numero di celle
`NUM_MCS`	Numero di passi Monte Carlo
`T`	Temperatura

Output:

`lattice_history`	Storico degli stati del reticolo
`chemical_history`	Storico degli stati del campo chimico

Function *main*:
- **STEP 0: Inizializzazione**
- Inizializza il reticolo e il campo chimico (*lattice*, *chemical*)
- Alloca memoria per i campi elettrici (E_field_x, E_field_y)
- Inizializza i volumi delle celle
- **STEP 1: Impostazione dell'ambiente**
- Inizializza le celle nel reticolo
- Imposta lo stato iniziale del reticolo (L^0)
- **STEP 2: Ciclo principale della simulazione**
- **for** $mcs = 1, \ldots, NUM\_MCS$ **do**
  - **if** $mcs \bmod 10 = 0$ **then**
    - Salva lo stato corrente del reticolo e dei campi chimici in history
  - **foreach** *pixel nel reticolo* **do**
    - Seleziona casualmente un sito P
    - Seleziona casualmente un vicino di P, Q
    - Calcola ΔH_{total} includendo i contributi di $H_{adhesion}$, H_{volume}, $H_{surface}$, $H_{chemotaxis}$ e $H_{electric}$
    - Applica l'algoritmo di Metropolis con probabilità basata su ΔH_{total} e T
    - **if** *la transizione è accettata* **then**
      - Aggiorna il reticolo e i volumi delle celle
  - Aggiorna il campo elettrico
  - Aggiorna il campo chimico e normalizza
- **STEP 3: Finalizzazione**
- Salva lo stato finale di *lattice* e *chemical* in *lattice_history* e *chemical_history*

Spiegazione dello pseudocodice MATLAB

La logica è suddivisa in tre fasi principali: inizializzazione, impostazione dell'ambiente, e ciclo principale della simulazione.

- **STEP 0**: Inizializzazione
  Nella fase di inizializzazione, vengono preparate le strutture di base necessarie

per la simulazione. In particolare, vengono creati il reticolo e il campo chimico come matrici bidimensionali, ed i campi elettrici (E_field_x e E_field_y). Inoltre, i volumi delle singole celle vengono inizializzati, per tenere traccia delle dimensioni delle cellule durante la simulazione.

- **STEP 1**: Impostazione dell'ambiente
  In questa fase vengono posizionate le cellule all'interno del reticolo. Ogni cellula riceve un identificatore univoco, e viene definito lo stato iniziale del reticolo (L^0), che rappresenta la configurazione di partenza per la simulazione.

- **STEP 2**: Ciclo principale della simulazione (Passi di Monte Carlo)
  La fase più importante del modello è il ciclo Monte Carlo. Ogni iterazione (*mcs*) rappresenta un passo di Monte Carlo e coinvolge l'interazione di tutti i pixel del reticolo. Durante ciascun passo, per ogni pixel viene scelto casualmente un pixel "sorgente" (P) e uno "target" (Q), che è un vicino di P. Si calcola il cambiamento di energia (ΔH_{total}) che risulterebbe dal passaggio di P verso Q, includendo i contributi energetici dati dall'adesione, dal volume, dalla superficie, dalla chemiotassi e dal campo elettrico. In seguito, si applica l'algoritmo di Metropolis, che valuta la probabilità di accettare il movimento basandosi su ΔH_{total} e sulla temperatura T del sistema. Se la transizione è accettata, il reticolo viene aggiornato e così i volumi delle celle coinvolte. Infine, vengono aggiornati sia il campo elettrico sia il campo chimico.

- **STEP 3**: Finalizzazione
  Al termine del ciclo Monte Carlo, il sistema registra lo stato finale del reticolo e del campo chimico, salvandoli nelle variabili $lattice\_history$ e $chemical\_history$ per una successiva analisi o visualizzazione.

Questo pseudocodice rappresenta il flusso di lavoro sequenziale utilizzato in MATLAB per simulare il comportamento cellulare. La sua struttura evidenzia chiaramente le fasi di inizializzazione, il ciclo Monte Carlo che regola le dinamiche del sistema e la fase di finalizzazione, utile per raccogliere i risultati. La natura iterativa e casuale della selezione dei pixel e delle transizioni rende il modello adeguato per simulare l'evoluzione delle cellule in un ambiente complesso, tenendo conto di molteplici contributi energetici che influenzano il comportamento cellulare.

Limiti della soluzione originale: performance e necessità di miglioramento

La versione iniziale del modello di simulazione implementato in MATLAB ha mostrato diversi limiti significativi, soprattutto in termini di prestazioni. MATLAB è uno strumento potente per lo sviluppo e la prototipazione rapida di algoritmi, ma presenta limitazioni evidenti quando viene utilizzato per simulazioni su larga scala, specialmente per la gestione di reticoli di grandi dimensioni e per la complessità computazionale crescente.

Uno dei principali problemi riscontrati riguarda la lentezza del ciclo Monte Carlo, dovuta al fatto che ogni aggiornamento del reticolo e la valutazione dei cambiamenti energetici sono eseguiti in maniera sequenziale e non ottimizzata. Questo approccio risulta inefficiente per sistemi che coinvolgono un elevato numero di celle e iterazioni, come nel caso della simulazione dell'angiogenesi, in cui è necessario simulare dinamiche cellulari su scala molto ampia. L'assenza di parallelizzazione ha quindi comportato

tempi di esecuzione troppo lunghi, limitando la possibilità di eseguire simulazioni con un numero elevato di passi Monte Carlo o dimensioni reticolari maggiori.

Un ulteriore limite significativo è legato alle routine di visualizzazione presenti nella versione originale del codice. Le funzionalità di visualizzazione in MATLAB, pur essendo convenienti per la prototipazione, hanno mostrato performance insoddisfacenti quando applicate a simulazioni di grandi dimensioni. La lentezza delle operazioni di rendering e l'incapacità di gestire efficacemente grandi quantità di dati visuali hanno reso difficile ottenere una rappresentazione interattiva e fluida dell'evoluzione del sistema. Ciò ha motivato la necessità di implementare un'alternativa più efficiente, portando alla successiva conversione delle routine di visualizzazione in C++ utilizzando la libreria OpenCV.

Inoltre, la dipendenza di MATLAB da librerie interne e il suo approccio interpretato hanno imposto ulteriori limiti sulle possibilità di ottimizzazione fine e di sfruttamento dell'hardware in maniera efficiente. La necessità di migliorare le prestazioni del modello ha quindi richiesto una riprogettazione, che prevedeva il passaggio a un linguaggio di programmazione più a basso livello come C, per permettere l'ottimizzazione delle operazioni a livello di memoria (ricorrendo anche all'uso di puntatori) e successivamente l'uso di tecniche di parallelizzazione tramite CUDA.

In definitiva, la versione in MATLAB ha rappresentato un'ottima base di partenza per la validazione del modello e la comprensione della dinamica cellulare, ma i limiti in termini di velocità di esecuzione, efficienza di visualizzazione e capacità di scalare verso simulazioni più complesse hanno reso necessaria una completa revisione dell'implementazione. Questo percorso evolutivo è stato necessario per garantire che il modello potesse essere utilizzato per simulazioni su larga scala in tempi ragionevoli, sfruttando al massimo le risorse computazionali disponibili.

2.2 Porting in C: una base per l'ottimizzazione

Dopo aver analizzato i limiti di performance della versione iniziale in MATLAB, si è deciso di effettuare un porting del codice in C, con l'obiettivo di migliorare l'efficienza computazionale e creare una base solida per una successiva parallelizzazione. L'uso del linguaggio C, noto per le sue capacità di ottimizzazione a basso livello e la sua efficienza nella gestione della memoria, ha permesso di affrontare i problemi legati ai lunghi tempi di esecuzione e all'inefficienza nel gestire grandi quantità di dati.

Il porting a C ha richiesto un ripensamento della struttura del codice per adattarlo alle peculiarità del linguaggio. A differenza di MATLAB, che è interpretato e orientato al calcolo matriciale, C consente una gestione diretta della memoria, permettendo un controllo dettagliato sull'allocazione e la deallocazione delle risorse. Questa caratteristica è stata sfruttata per migliorare la gestione delle matrici e delle strutture dati del modello, rendendo il codice significativamente più efficiente. L'inizializzazione del reticolo, il calcolo dei cambiamenti energetici, e l'applicazione dell'algoritmo di Metropolis sono stati riscritti per sfruttare al meglio le capacità di elaborazione del linguaggio C.

Un altro vantaggio derivante dall'uso di C è stato la possibilità di preparare il codice per la parallelizzazione. Essendo C un linguaggio compilato e molto vicino all'hardware, la sua compatibilità con le tecnologie di parallel computing, come CUDA, ha aperto la strada per implementare versioni ottimizzate del modello. Ogni funzione del

codice MATLAB è stata gradualmente convertita, mantenendo la correttezza dell'algoritmo originale e verificando attentamente i risultati, in modo da preservare l'integrità del modello di simulazione. Questo processo di conversione progressiva ha permesso di testare ogni parte del codice e di identificare aree critiche su cui intervenire per una successiva fase di ottimizzazione parallela.

Il passaggio a C ha anche permesso di migliorare le performance delle routine di visualizzazione, che nella versione MATLAB risultavano lente e non adatte a simulazioni su larga scala. Il porting delle routine di visualizzazione in C++ utilizzando la libreria OpenCV ha consentito di ottenere una rappresentazione dei dati molto più efficiente, riducendo i tempi di rendering e migliorando l'interattività. OpenCV, grazie alle sue ottimizzazioni grafiche, ha reso possibile visualizzare l'evoluzione del sistema in maniera fluida, anche quando la complessità del reticolo e del campo chimico aumentavano.

Per concludere, il porting del modello a C ha rappresentato un passaggio cruciale per migliorare l'efficienza computazionale e per predisporre il codice alla parallelizzazione. L'utilizzo di C ha consentito di sfruttare al massimo l'hardware disponibile e di porre le basi per una versione ottimizzata in CUDA, con l'obiettivo di supportare simulazioni su larga scala, migliorando sia i tempi di esecuzione sia la qualità della visualizzazione.

2.2.1 Uso di LAPACK e SuiteSparse per la risoluzione di sistemi lineari

Nel porting del modello di simulazione in C, uno dei principali miglioramenti rispetto alla versione in MATLAB riguarda la gestione della risoluzione dei sistemi lineari, fondamentale per il calcolo dei campi elettrici e per determinare la dinamica del sistema. A tal fine, sono state utilizzate le librerie LAPACK e SuiteSparse, entrambe progettate per risolvere problemi algebrici in maniera efficiente.

Costruzione della matrice e del vettore

Il processo, indipendentemente dall'uso di LAPACK o SuiteSparse, inizia con la costruzione della matrice A e del vettore b, dove la matrice A è una rappresentazione numerica del reticolo e dei collegamenti tra i suoi nodi. Nello specifico:

- **Allocazione della matrice e del vettore**: La matrice A viene mappata come un vettore unidimensionale di dimensione $n \times n$, dove n è il numero totale di elementi nel reticolo ($n = \texttt{size} \times \texttt{size}$). Allo stesso modo, il vettore b viene allocato con n elementi. L'allocazione di A e b viene fatta inizialmente con tutti i valori impostati a zero.

- **Costruzione della matrice**: La matrice A viene popolata con valori diagonali e valori che rappresentano le connessioni con i vicini. In particolare, per ciascun elemento i, j della matrice, viene assegnato un valore diagonale pari a 4.0 e un valore di -1.0 per ciascun vicino (sopra, sotto, a destra e a sinistra). Questi valori rappresentano un modello numerico simile a un operatore laplaciano, utilizzato per descrivere le interazioni tra i punti del reticolo.

- **Costruzione del vettore** b: Il vettore b viene popolato utilizzando i valori del campo chimico (`chemical[i][j]`). Ogni elemento del reticolo contribuisce al vettore b con il suo valore chimico corrente.

Fattorizzazione LU: concetti generali

La fattorizzazione LU è un metodo di decomposizione di una matrice quadrata A in due matrici triangolari: una matrice triangolare inferiore L e una matrice triangolare superiore U, tale che $A = L \cdot U$. Questo metodo è fondamentale per risolvere sistemi lineari, calcolare determinanti e trovare inverse di matrici in modo efficiente. In particolare, la fattorizzazione LU permette di ridurre un problema complesso di risoluzione di sistemi lineari in due problemi più semplici, che coinvolgono matrici triangolari, rendendo le operazioni numeriche più stabili e veloci.

La matrice triangolare inferiore L ha elementi diversi da zero solo sotto la diagonale principale, mentre la matrice triangolare superiore U ha elementi diversi da zero solo sulla diagonale e sopra di essa. Il vantaggio principale di questa fattorizzazione è la semplificazione delle operazioni di risoluzione dei sistemi lineari, poiché le equazioni con matrici triangolari possono essere risolte in modo diretto utilizzando la sostituzione in avanti (per L) e la sostituzione all'indietro (per U).

Pivoting e stabilità numerica

Uno dei problemi principali nella fattorizzazione LU è la possibile instabilità numerica [19] (*Gene H Golub e Charles F Van Loan*, *2013*). Se durante la decomposizione il valore di un elemento diagonale è molto piccolo rispetto agli altri elementi, i calcoli possono diventare numericamente instabili, generando errori significativi. Per ovviare a questo problema, viene utilizzata la tecnica del pivoting parziale.

Il pivoting parziale consiste nel riordinare le righe della matrice A in modo che l'elemento diagonale più grande sia posto nella posizione corrente della diagonale prima di procedere alla decomposizione [50] (*Lloyd N Trefethen e David Bau III*, *1997*). Questo riordinamento riduce il rischio di instabilità numerica migliorando la precisione dei risultati. Nel contesto del codice fornito, il pivoting parziale è implementato nella funzione `LAPACKE_dgesv` utilizzata con LAPACK, che applica automaticamente il pivoting durante la fattorizzazione LU per garantire stabilità.

Nel caso di SuiteSparse, il pivoting è gestito attraverso le funzioni di permutazione durante la fase di risoluzione del sistema. La funzione `cs_ipvec` applica una permutazione degli elementi per garantire una corretta esecuzione dei passi di fattorizzazione e risoluzione.

Fattorizzazione simbolica e numerica

Un aspetto rilevante della fattorizzazione di una matrice sparsa è la distinzione tra fattorizzazione simbolica e fattorizzazione numerica:

- **Fattorizzazione simbolica**: Questa fase riguarda l'analisi della struttura della matrice A per determinare la posizione degli elementi non nulli nelle matrici L e U senza effettivamente calcolare i loro valori. La fattorizzazione simbolica è importante per ottimizzare l'allocazione della memoria e minimizzare il numero

di operazioni necessarie durante la successiva fase numerica. Nel caso di SuiteSparse, la fattorizzazione simbolica è eseguita tramite la funzione `cs_sqr`, che analizza la struttura della matrice e decide l'ordine delle operazioni di decomposizione. Questo è fondamentale per le matrici sparse, dove la gestione efficiente della struttura è cruciale per le prestazioni.

- **Fattorizzazione numerica**: Questa fase viene eseguita dopo la fattorizzazione simbolica e consiste nel calcolo effettivo dei valori delle matrici L e U. Nel contesto di SuiteSparse, questa fase è implementata dalla funzione `cs_lu`, che esegue la decomposizione utilizzando la struttura determinata dalla fase simbolica. Nel caso di LAPACK, non viene effettuata una distinzione esplicita tra la fase simbolica e quella numerica, poiché la funzione `LAPACKE_dgesv` calcola direttamente sia la struttura che i valori delle matrici L e U.

Utilizzo nella risoluzione di sistemi lineari

La fattorizzazione LU è utilizzata sia in LAPACK che in SuiteSparse per risolvere il sistema lineare $A \cdot x = b$. Nel caso di LAPACK, la funzione `LAPACKE_dgesv` fattorizza la matrice A e utilizza le matrici L e U per risolvere il sistema in due passaggi: risoluzione di $L \cdot y = b$ tramite sostituzione in avanti e successivamente $U \cdot x = y$ tramite sostituzione all'indietro. SuiteSparse, d'altra parte, utilizza una combinazione di fattorizzazione simbolica e numerica per sfruttare la struttura sparsa della matrice e risolvere il sistema in modo efficiente, riducendo il numero di operazioni e l'utilizzo di memoria. La fattorizzazione LU rappresenta un passo cruciale nella risoluzione di sistemi lineari complessi, sia in contesti di matrici dense che sparse. L'uso del pivoting parziale in LAPACK e la combinazione di fattorizzazione simbolica e numerica in SuiteSparse garantiscono la stabilità e l'efficienza necessarie per affrontare le simulazioni complesse descritte in questo lavoro.

Calcolo dei campi elettrici

Dopo aver risolto il sistema lineare, la soluzione x viene utilizzata per calcolare i campi elettrici E_{field_x} e E_{field_y}:

Calcolo delle componenti del campo elettrico: I campi elettrici vengono calcolati utilizzando un'approssimazione delle differenze finite centralizzate. In particolare, per ciascun punto del reticolo, il valore del campo E_{field_x} viene calcolato come la differenza tra i valori della soluzione b nei punti adiacenti lungo l'asse x, e similmente per E_{field_y} lungo l'asse y. Questa approssimazione consente di stimare il gradiente della soluzione, che rappresenta l'intensità e la direzione del campo elettrico in ogni punto del reticolo.

LAPACK: risoluzione di sistemi con matrici dense

La libreria LAPACK (*Linear Algebra PACKage*, in particolare `lapacke.h`) è stata utilizzata per la risoluzione di sistemi lineari che coinvolgono matrici dense, ovvero matrici in cui la maggior parte degli elementi è non nulla. Nello specifico, è stata impiegata la funzione `LAPACKE_dgesv`, che è in grado di risolvere un sistema lineare del tipo $A \cdot x = b$ utilizzando la fattorizzazione LU con pivoting parziale [34] (*Netlib LAPACK Project, 2024*). Questa funzione calcola una decomposizione della matrice

A in due matrici triangolari (una triangolare inferiore e una triangolare superiore), che viene poi utilizzata per risolvere il sistema lineare in maniera efficiente.

L'uso di `LAPACKE_dgesv` è particolarmente indicato per sistemi in cui la matrice A è densa e la risoluzione diretta è più vantaggiosa rispetto a metodi iterativi. Tuttavia, quando la dimensione del reticolo aumenta, l'utilizzo di LAPACK con matrici dense può risultare estremamente limitante a causa dei requisiti di memoria elevati e della complessità computazionale $O(n^3)$ [33] (*Netlib LAPACK Project*, *2024*). Questo ha motivato l'adozione di un approccio alternativo per sistemi più grandi e più sparsi, portando all'utilizzo della libreria SuiteSparse. In particolare, una volta costruiti la matrice A e il vettore b, viene utilizzata la funzione `LAPACKE_dgesv` per risolvere il sistema lineare:

- **LAPACKE_dgesv**: La funzione `LAPACKE_dgesv` è la funzione principale utilizzata per risolvere il sistema lineare $A \cdot x = b$. Questa funzione utilizza la fattorizzazione LU della matrice A, ovvero scompone A in una matrice triangolare inferiore L e una matrice triangolare superiore U, tale che $A = L \cdot U$. Il pivoting parziale viene applicato per ridurre il rischio di instabilità numerica, riordinando le righe della matrice durante la fattorizzazione in base ai valori assoluti massimi.

- **Gestione degli errori**: Dopo la chiamata a `LAPACKE_dgesv`, il codice verifica il valore restituito da `info` per determinare se la risoluzione del sistema è stata eseguita con successo. Se `info` è pari a 0, il sistema è stato risolto correttamente; valori diversi da 0 indicano errori, come un parametro non valido o una matrice singolare (che non ha una soluzione unica).

Vantaggi dell'utilizzo di LAPACK

L'uso di LAPACK per risolvere sistemi con matrici dense ha fornito un metodo efficiente per gestire reticoli di dimensioni ridotte, dove l'uso di matrici dense non rappresenta un problema significativo in termini di consumo di memoria. La fattorizzazione LU è un metodo robusto che garantisce soluzioni accurate per i sistemi lineari, rendendolo una scelta appropriata per simulazioni su scala media. Tuttavia, per reticoli più grandi, la complessità $O(n^3)$ della fattorizzazione LU e i requisiti di memoria elevati rendono questo approccio meno adatto rispetto a una soluzione basata su matrici sparse, come quella implementata con SuiteSparse.

SuiteSparse: gestione delle matrici sparse

Nel modello C, la libreria SuiteSparse è stata utilizzata per gestire e risolvere sistemi lineari con matrici sparse. SuiteSparse offre strumenti ottimizzati per la gestione delle matrici sparse, che si adattano perfettamente alle caratteristiche del problema, poiché le matrici che descrivono il reticolo sono prevalentemente popolate da valori nulli [48] (*Timothy A. Davis*, *2024*). Nel porting a C, è stato implementato un sistema che utilizza la rappresentazione della matrice nel formato *"triplet"* (o anche *Coordinate List*, COO) per costruire la matrice sparsa A, successivamente convertita nel formato *Compressed-Column Storage* (CSC) per ottimizzare le operazioni di risoluzione. Il formato *triplet* è una rappresentazione delle matrici sparse in cui ogni elemento non nullo è memorizzato come una tripla (i, j, v), dove i e j indicano la riga e la colonna dell'elemento, e v è il valore. Questo formato è conveniente per costruire e modificare

la matrice elemento per elemento, prima di convertirla in una rappresentazione più compatta come il formato CSC. Il formato CSC è una rappresentazione efficiente delle matrici sparse, in cui vengono memorizzati solo i valori non nulli insieme agli indici di riga e ai puntatori di colonna, riducendo così il consumo di memoria e permettendo operazioni più rapide sulle colonne della matrice.

L'approccio utilizzato nel codice sfrutta diverse funzioni della libreria CSparse (`cs.h` che è una parte della libreria SuiteSparse):

- `cs_spalloc`: Questa funzione è utilizzata per allocare una matrice sparsa A inizialmente nel formato *triplet*. Questo formato è conveniente per costruire la matrice elemento per elemento, in quanto consente di aggiungere direttamente valori e indici di riga e colonna. Nel codice, la matrice A rappresenta la discretizzazione del reticolo con connessioni ai vicini, un modello simile a un operatore laplaciano.

- `cs_entry`: La funzione viene utilizzata per inserire gli elementi nella matrice *triplet*. In particolare, per ciascun *pixel* del reticolo viene impostato il valore diagonale e i valori delle connessioni con i vicini (ad esempio, le celle adiacenti sopra, sotto, a destra e a sinistra). Questi valori rappresentano il contributo energetico del campo elettrico e chimico associato a ciascun punto del reticolo.

- `cs_compress`: Una volta costruita la matrice A nel formato *triplet*, viene convertita nel formato *compress-column* (CSC). Questo formato è più compatto e consente un accesso più efficiente durante la risoluzione del sistema lineare, riducendo il costo di memorizzazione e di calcolo per le operazioni successive.

La risoluzione del sistema lineare avviene poi tramite una sequenza di passi che coinvolge la fattorizzazione LU:

- `cs_sqr`: Viene eseguita una fattorizzazione simbolica della matrice utilizzando la funzione `cs_sqr`, la quale analizza la struttura della matrice e prepara la fase successiva. Questa funzione determina l'ordine in cui eseguire le operazioni di fattorizzazione e serve a ridurre il riempimento nella matrice durante la decomposizione, migliorando quindi l'efficienza computazionale.

- `cs_lu`: Successivamente, la fattorizzazione numerica viene eseguita con `cs_lu`, che utilizza la fattorizzazione LU per scomporre la matrice A nelle componenti triangolari L (inferiore) e U (superiore). Questo approccio è particolarmente utile per le matrici sparse che non sono necessariamente simmetriche definite positive, consentendo una maggiore flessibilità nell'affrontare vari tipi di problemi.

- `cs_ipvec`, `cs_lsolve`, e `cs_usolve`: La soluzione del sistema si completa risolvendo le equazioni $L * y = P * b$ e $U * x = y$ con le funzioni `cs_lsolve` e `cs_usolve`, rispettivamente. La funzione `cs_ipvec` viene utilizzata per applicare la permutazione appropriata al vettore b e al risultato, garantendo che l'ordine degli elementi sia corretto per la soluzione finale.

Questo processo di fattorizzazione e risoluzione è stato utilizzato per risolvere il sistema lineare che descrive il campo chimico e il campo elettrico nel reticolo. La soluzione ottenuta viene poi utilizzata per calcolare i campi E_{field_x} e E_{field_y} tramite l'approssimazione delle differenze finite centralizzate, che permette di stimare i gradienti necessari a modellare l'effetto del campo elettrico sulle cellule.

Vantaggi dell'utilizzo di SuiteSparse

L'uso di SuiteSparse per la risoluzione dei sistemi con matrici sparse ha portato a notevoli vantaggi in termini di efficienza e riduzione dei requisiti di memoria. La rappresentazione sparsa delle matrici ha ridotto la quantità di memoria necessaria per memorizzare e manipolare il sistema, mentre la fattorizzazione LU ha garantito una risoluzione accurata e relativamente efficiente del sistema lineare. Questo approccio ha reso possibile l'implementazione su reticoli di dimensioni maggiori, mantenendo la scalabilità del modello e preparandolo per l'ottimizzazione successiva tramite l'uso di CUDA.

L'integrazione di LAPACK per le matrici dense e SuiteSparse per le matrici sparse ha permesso di ottenere un codice flessibile ed efficiente, capace di adattarsi a diverse tipologie di reticoli e condizioni del modello, garantendo una gestione ottimale delle risorse e una riduzione dei tempi di calcolo.

2.2.2 Descrizione dell'implementazione in C

Il porting del codice MATLAB in C ha comportato una completa riscrittura del modello, con un'attenzione particolare all'efficienza computazionale e alla gestione della memoria. L'implementazione è stata suddivisa in più file per garantire una struttura modulare e più facile da manutenere. In particolare, sono stati utilizzati tre file principali: `cpm.c`, `cpm_functions.c`, e `cpm.h`, ognuno dei quali contribuisce in modo distinto alla funzionalità del modello.

Il file `cpm.c` rappresenta il punto di ingresso dell'applicazione e contiene il ciclo principale della simulazione di Monte Carlo. Al suo interno troviamo la gestione dei parametri di input e l'inizializzazione delle strutture dati principali, come il reticolo (`lattice`) e il campo chimico (`chemical`). La simulazione si svolge iterando attraverso un ciclo di passi Monte Carlo (MCS), con ogni passaggio che comporta la selezione casuale di un sito del reticolo e l'applicazione dell'algoritmo di Metropolis per decidere se accettare una potenziale transizione. Questo file contiene anche le funzioni per salvare lo stato delle matrici per la visualizzazione, esattamente come fatto in MATLAB, con lo scopo di mantenere traccia dell'evoluzione del sistema per eventuali analisi successive o visualizzazioni.

Il file `cpm_functions.c` contiene l'implementazione delle varie funzioni utilizzate dal programma. Qui si trovano le routine per la risoluzione dei sistemi lineari, che rappresentano uno dei componenti fondamentali dell'implementazione in C. In particolare, è presente una funzione per risolvere sistemi lineari utilizzando le librerie LAPACK per matrici dense, e una funzione che fa uso della libreria SuiteSparse per la risoluzione di sistemi con matrici sparse. Quest'ultima scelta permette di migliorare notevolmente l'efficienza computazionale, specialmente quando si opera su grandi reticoli con molti elementi nulli. Il file `cpm.h` definisce i prototipi delle funzioni e le costanti principali utilizzate nel modello. Ad esempio, definisce parametri come la dimensione del reticolo (`LATTICE_SIZE`), il numero di cellule (`NUM_CELLS`) e il numero di passi Monte Carlo (`NUM_MCS`). Questo file include anche i parametri di controllo dell'energia, come `LAMBDA_V`, `LAMBDA_S`, `LAMBDA_C` e `LAMBDA_E`, che determinano l'importanza relativa dei diversi contributi energetici nel calcolo del cambiamento di energia per ogni possibile transizione del reticolo. Questi parametri sono fondamentali per il controllo della dinamica del modello e per l'adattamento delle simulazioni ai vari scenari biologici studiati. La versione C del codice ha mantenuto la logica generale dell'algoritmo di

Monte Carlo del codice MATLAB, ma è stata ottimizzata per migliorare l'efficienza e ridurre i tempi di esecuzione. Grazie all'uso delle librerie LAPACK e SuiteSparse, l'implementazione in C ha migliorato la gestione dei calcoli matriciali, con una particolare enfasi sulla risoluzione efficiente dei sistemi lineari, che sono cruciali per il calcolo dei campi elettrici. Inoltre, è stata aggiunta una nuova sezione per leggere e visualizzare i dati scritti dal codice in C, consentendo una visualizzazione più efficiente rispetto a quella effettuata direttamente in MATLAB. Questa implementazione modulare ha fornito una base robusta per il passaggio successivo: la conversione in CUDA, che ha ulteriormente accelerato il modello, rendendolo adatto a simulazioni su larga scala e a scenari più complessi. Il passaggio a C non solo ha permesso di ottenere miglioramenti significativi nelle prestazioni, ma ha anche reso il codice più flessibile per ulteriori ottimizzazioni e adattamenti a diverse architetture di calcolo.

2.2.3 Pseudocodice sequenziale in C

Algoritmo 2: Modello Cellulare di Potts (C)

Input:

`LATTICE_SIZE`	Dimensione del reticolo
`NUM_CELLS`	Numero di celle
`NUM_MCS`	Numero di passi Monte Carlo
`T`	Temperatura

Output:

`lattice_history`	Storico degli stati del reticolo
`chemical_history`	Storico degli stati del campo chimico

```
1  Function main:
2      STEP 0: Inizializzazione
3      Inizializza il reticolo e i campi chimici
4      Alloca la memoria per i campi elettrici e le variabili di simulazione
5      STEP 1: Imposta l'ambiente
6      Inizializza lo stato iniziale del reticolo L^0
7      Inizializza le celle all'interno del reticolo
8      STEP 2: Ciclo principale della simulazione (passi di Monte Carlo)
9      for mcs = 1,...,NUM_MCS do
10         if mcs mod 10 = 0 then
11             Salva lo stato corrente del reticolo e dei campi chimici in history
12         foreach pixel nel reticolo do
13             Seleziona un pixel casuale P sul reticolo come "sorgente"
14             Seleziona un vicino casuale Q di P come "target"
15             Calcola ΔH_total includendo i contributi di H_adhesion, H_volume,
                H_surface, H_chemotaxis e H_electric
16             Applica l'algoritmo di Metropolis con probabilità basata su ΔH_total
                e T
17             if la transizione è accettata then
18                 Aggiorna il reticolo e i volumi delle celle
19         Risolvi il sistema per il campo elettrico utilizzando un metodo denso o
            sparso
20         Aggiorna i campi chimici e normalizza
21     STEP 3: Finalizzazione
22     Dealloca la memoria utilizzata
```

Spiegazione dello pseudocodice

Lo pseudocodice proposto per la versione sequenziale del Modello Cellulare di Potts (CPM) descrive il flusso del programma implementato in C, evidenziando i principali passi del ciclo di simulazione Monte Carlo. Ogni step dell'algoritmo è progettato per catturare la dinamica cellulare su un reticolo, aggiornando le proprietà energetiche e chimiche per rappresentare il comportamento delle cellule nel tempo. Di seguito, una spiegazione dettagliata di ciascuno degli step:

1. **STEP 0: Inizializzazione** In questa fase iniziale vengono definiti i parametri principali del reticolo e delle celle:

   - **Reticolo e campi chimici**: Vengono inizializzati il reticolo, che rappresenta la disposizione spaziale delle celle, e i campi chimici associati ad esso.
   - **Allocazione memoria**: Vengono allocate le strutture di memoria necessarie per i campi elettrici e altre variabili di simulazione, preparando l'ambiente per i calcoli successivi.

2. **STEP 1: Imposta l'ambiente** Una volta che la memoria è stata allocata, l'algoritmo imposta l'ambiente iniziale:

   - **Stato iniziale del reticolo**: Viene assegnato lo stato iniziale del reticolo (L^0), che rappresenta la configurazione di partenza delle celle.
   - **Inizializzazione delle celle**: Ogni cella viene inizializzata con i parametri di volume target e con la disposizione nel reticolo.

3. **STEP 2: Ciclo principale della simulazione** Questo step rappresenta il ciclo di Monte Carlo che consente di simulare l'evoluzione temporale del sistema:

   - **Passo Monte Carlo (MCS)**: Per ogni passo di Monte Carlo (`mcs`), l'algoritmo simula l'evoluzione del reticolo.
   - **Salvataggio degli stati**: Se il valore di `mcs` è multiplo di 10, lo stato corrente del reticolo e dei campi chimici viene salvato per un'eventuale visualizzazione futura.
   - **Aggiornamento per ogni pixel**:
     – Viene selezionato casualmente un pixel P nel reticolo come "sorgente" e uno dei suoi vicini Q come "target".
     – Si calcola ΔH_{total}, che rappresenta il cambiamento energetico derivante dal possibile trasferimento di P su Q, includendo vari contributi energetici: $H_{adhesion}$ (adesione), H_{volume} (volume), $H_{surface}$ (superficie), $H_{chemotaxis}$ (chemiotassi), e $H_{electric}$ (campo elettrico).
   - **Algoritmo di Metropolis**: Viene applicato l'algoritmo di Metropolis per determinare se la transizione sarà accettata o meno, in base al cambiamento di energia ΔH_{total} e alla temperatura T. Se la transizione viene accettata, si aggiorna il reticolo e il volume delle celle coinvolte.
   - **Risoluzione del sistema per il campo elettrico**: Alla fine di ogni ciclo di Monte Carlo, viene risolto un sistema lineare per determinare i nuovi campi elettrici. Questo sistema viene risolto utilizzando una tecnica basata su matrici dense o sparse, in funzione della rappresentazione più efficiente per il problema in esame.
   - **Aggiornamento dei campi chimici**: I campi chimici vengono aggiornati e normalizzati, per garantire che le concentrazioni rientrino nei limiti desiderati.

4. **STEP 3: Finalizzazione**

Deallocazione della memoria: Tutte le strutture di memoria allocate durante i passi precedenti vengono deallocate per evitare perdite di memoria. Questo è particolarmente importante in C, dove la gestione della memoria è esplicita.

Questo pseudocodice evidenzia in maniera chiara i passi fondamentali del ciclo di simulazione Monte Carlo nella versione sequenziale del modello in C. La struttura del programma è pensata per simulare accuratamente la dinamica delle cellule, consentendo un'estensione successiva per l'ottimizzazione e parallelizzazione con CUDA.

2.2.4 Generazione di file per la visualizzazione dei risultati

Durante la simulazione del Modello Cellulare di Potts (CPM), è fondamentale poter visualizzare l'evoluzione del sistema per comprendere meglio il comportamento delle celle e validare i risultati ottenuti. A tal fine, è stata implementata una funzione in C denominata `write_matrix`, che consente di salvare gli stati intermedi del reticolo (`lattice`) e dei campi chimici (`chemical`) su file.

La funzione `write_matrix` genera due file di testo: `lattice.txt` e `chemical.txt`, contenenti rispettivamente la storia del reticolo e delle concentrazioni chimiche. Ogni file riporta una sequenza di matrici che rappresentano gli stati del sistema a intervalli specifici, in modo da fornire una base per la visualizzazione temporale dell'evoluzione della simulazione. La funzione crea una rappresentazione leggibile di ciascuna matrice, con ogni elemento separato da uno spazio, e con le matrici temporali numerate e separate per garantire una facile analisi successiva.

Nella fase iniziale del progetto, questi file sono stati creati principalmente per poter essere letti all'interno dell'ambiente MATLAB, sfruttando le routine di visualizzazione già presenti nel codice originale ricevuto. Per questa ragione, è stata introdotta una modifica nel file `cpm_angio_2.m` in MATLAB per consentire la lettura di `lattice.txt` e `chemical.txt`, e utilizzare così le funzioni di visualizzazione già esistenti. Questa soluzione ha permesso di effettuare una prima validazione dei risultati simulati in C, mantenendo le stesse modalità di visualizzazione utilizzate nella versione originale in MATLAB.

Successivamente, tuttavia, è stato necessario sviluppare una soluzione più efficiente per la visualizzazione, in particolare per supportare le simulazioni su larga scala e superare i limiti di performance di MATLAB. A tal fine, è stata realizzata una routine di visualizzazione in C++ utilizzando la libreria OpenCV. Anche in questo caso, i file `lattice.txt` e `chemical.txt` generati dal codice C sono stati utilizzati come input per la routine di visualizzazione, fornendo continuità tra le diverse fasi del progetto. L'uso di OpenCV ha reso possibile una gestione più efficiente della visualizzazione, sia in termini di tempo di elaborazione sia di rappresentazione grafica.

In conclusione, la generazione di file per la visualizzazione dei risultati è stata un passo cruciale nel processo di validazione e sviluppo del modello. Ha fornito la flessibilità necessaria per connettere diverse fasi del progetto, dalla prima implementazione in MATLAB alla visualizzazione avanzata con OpenCV, garantendo coerenza nell'analisi dei risultati e migliorando le capacità di interpretazione dei fenomeni simulati.

2.3 Visualizzazione in C++ e OpenCV

L'evoluzione del codice di simulazione ha portato, oltre all'ottimizzazione dell'algoritmo di calcolo, anche a un significativo miglioramento delle routine di visualizzazione. Nella versione originale scritta in MATLAB, la rappresentazione grafica dei risultati era un processo fondamentale per l'analisi del comportamento del modello, ma risultava limitata in termini di performance e flessibilità. L'elaborazione e il rendering dei dati in MATLAB erano lenti, soprattutto per simulazioni su larga scala, e questo rappresentava un collo di bottiglia nell'analisi dei risultati.

Per migliorare l'efficienza e garantire una visualizzazione più fluida, le routine di visualizzazione sono state portate in C++ utilizzando la libreria OpenCV, nota per le sue capacità avanzate di elaborazione delle immagini. Questa libreria ha consentito di gestire grandi quantità di dati in modo molto più efficiente, di generare visualizzazioni di qualità superiore e di produrre direttamente output video della simulazione.

L'approccio adottato ha previsto inizialmente il caricamento dei dati da file generati dal programma in C, per poi applicare tecniche avanzate di visualizzazione e creare video che rappresentano l'evoluzione del sistema. In particolare, l'utilizzo di OpenCV ha permesso di integrare annotazioni, legende e coordinate dettagliate nelle visualizzazioni, migliorando la comprensione dei fenomeni simulati. Questo ha reso il processo non solo più veloce, ma anche più interattivo e personalizzabile, risultando adatto sia per un'analisi scientifica approfondita che per una presentazione più chiara dei risultati.

2.3.1 OpenCV: funzioni e strutture dati utilizzate

Per la visualizzazione del modello di simulazione è stata usata la libreria OpenCV, che è una delle più popolari librerie open-source per l'elaborazione delle immagini [41] (*OpenCV Team*, *2024*). OpenCV ha permesso di implementare efficientemente la rappresentazione grafica del reticolo cellulare e del campo chimico, nonché di salvare l'evoluzione della simulazione in un video. Di seguito, vengono descritte le principali strutture dati e funzioni di OpenCV utilizzate per questo progetto.

Struttura dati `cv::Mat`:

In OpenCV le immagini ed i video sono rappresentati tramite un oggetto di tipo `cv::Mat` [40] (*OpenCV Team*, *2024*), che in questo caso è stato utilizzato per rappresentare le matrici per il campo chimico ed il reticolo della simulazione. Ogni `cv::Mat` contiene una matrice multidimensionale in cui sono memorizzati i dati in modo efficiente. Nel contesto della simulazione, i dati chimici sono memorizzati come matrici in formato `CV_32F` (floating point a 32 bit), mentre i dati del reticolo sono memorizzati in formato `CV_8UC1` (intero a 8 bit con un canale).

Funzioni di lettura e normalizzazione:

Le funzioni utilizzate per leggere i file contenenti i dati del reticolo e del campo chimico sono `loadAllFrames`, `readLatticeMatrix` e `readAndProcessChemicalMatrix`. In particolare, la funzione `readAndProcessChemicalMatrix` normalizza i dati chimici utilizzando `cv::normalize`, scalando i valori tra 0 e 255 per renderli visualizzabili, e poi li converte in un formato adatto per la visualizzazione (`CV_8UC1`).

Funzione di visualizzazione delle celle:

La funzione `visualize_cells` è responsabile del disegno del reticolo sul frame di visualizzazione. Questa funzione utilizza `cv::rectangle` che permette di tracciare un rettangolo per disegnare le celle del reticolo e i loro contorni sottili, in modo da distinguere chiaramente i bordi tra le celle adiacenti. Ogni cella ha un colore assegnato casualmente utilizzando `cv::RNG`, una classe per generare numeri casuali *(Random Number Generator)* e ricorrendo alla funzione `rng.uniform(a, b)` che restituisce valori casuali in un intervallo specificato.

Creazione di legenda (*Color Bar*):

Per rappresentare il campo chimico, è stata implementata una legenda utilizzando la funzione `createColorBar`. Questa funzione crea un gradiente verticale (da 0 a 255) per poi applicare una mappa di colori tramite `cv::applyColorMap` al fine di rendere visibile la concentrazione chimica nelle simulazioni. La funzione `addColorBarLabels` aggiunge etichette per mostrare i valori minimi e massimi associati alla color bar, utilizzando il modulo FreeType (`cv::freetype::FreeType2`) per una migliore qualità del testo.

Annotazioni sulle immagini:

Per migliorare la leggibilità delle visualizzazioni, sono state aggiunte annotazioni e coordinate tramite la funzione `drawCoordinatesOutside`. Questa funzione utilizza il modulo FreeType di OpenCV per aggiungere testo personalizzato (con un font esterno) e disegnare le tacche per le coordinate sui lati dell'immagine. Il modulo `freetype` consente di avere maggiore controllo sulla resa grafica del testo rispetto alla funzione predefinita `cv::putText`.

Gestione del video:

La funzione `visualizeInit` si occupa di gestire l'intera sequenza di visualizzazione e di salvare l'output come video utilizzando la classe `cv::VideoWriter`. Questa classe consente di salvare una serie di frame come video, nel formato H.264, utile per analisi successive o per presentare i risultati della simulazione.

2.3.2 Pseucodocodice della visualizzazione

Di seguito uno pseudocodice che descrive la logica usata per la funzione `visualizeInit`:

Algoritmo 3: Inizializzazione della visualizzazione (OpenCV)

Input:

`latticeFile`	File contenente i dati del reticolo
`chemicalFile`	File contenente i dati chimici
`num_frames`	Numero totale di frame da visualizzare
`desired_fps`	Frequenza dei frame desiderata per il video

Output:
Video della simulazione
visualizzazione dei frame

Function *visualizeInit*:
- **STEP 0: Caricamento delle librerie e configurazione del video**;
- Carica le librerie OpenCV e FreeType2 per la gestione dei font;
- Carica i file di lattice e chemical per la simulazione;
- Imposta i parametri per dimensioni, padding e configurazione del video writer;
- **STEP 1: Caricamento dei frame**;
- Carica tutti i frame per lattice e chemical tramite la funzione $loadAllFrames()$;
- **STEP 2: Ciclo di visualizzazione dei frame**;
- **for** $frame = 0, \ldots, num\_frames - 1$ **do**
  - Leggi il frame corrente del reticolo (*lattice*) e dei dati chimici (*chemical*);
  - Visualizza le celle del reticolo usando la funzione $visualize\_cells()$;
  - Applica una mappa di colori per visualizzare il campo chimico;
  - Ridimensiona le immagini per una maggiore risoluzione;
  - **STEP 3: Preparazione della composizione video** ;
  - Crea un'immagine di sfondo con padding per includere entrambi i frame;
  - Aggiungi titoli per le configurazioni cellulari e le concentrazioni chimiche;
  - Crea e posiziona una legenda per la barra dei colori usando la funzione $createColorBar()$;
  - Aggiungi le coordinate esterne e le etichette alla barra dei colori;
  - **STEP 4: Finalizzazione del frame**;
  - Copia le immagini del reticolo e del campo chimico nello sfondo;
  - Aggiungi il numero del frame corrente;
  - Scrivi il frame sul video output;
  - Visualizza i frame per il controllo durante l'esecuzione;
- **STEP 5: Rilascio della memoria** ;
- Rilascia il video writer e chiudi tutte le finestre di visualizzazione;

Spiegazione dello pseudocodice

La funzione `visualizeInit()` si occupa di gestire la visualizzazione della simulazione del CPM con il campo elettrico utilizzando OpenCV. A seguire, sono riportati i

passaggi principali:

1. **Caricamento del font personalizzato:**
   La funzione carica due font tramite `FreeType2`, come già spiegato precedentemente, permettendo di aggiungere testi personalizzati alle visualizzazioni.

2. **Configurazione del video writer:**
   Viene creato un oggetto `VideoWriter` che genera un file video utilizzando il codec H.264 (`cpm_simulation_opencv.mp4`). Questo video rappresenta una versione registrata della simulazione, utile per l'analisi successiva.

3. **Caricamento dei frame:**
   Tutti i frame di reticolo (`lattice`) e campo chimico (`chemical`) vengono caricati da due file di testo (`lattice.txt` e `chemical.txt`). La funzione che si occupa di leggere le matrici dai file e salvarle nelle rispettive strutture di dati per facilitarne la visualizzazione in sequenza è `loadAllFrames()` .

4. **Preparazione della visualizzazione dei frame:**
   Ogni frame del reticolo e del campo chimico viene processato tramite funzioni come `visualize_cells()` e `applyColorMap()`, che rappresentano rispettivamente la configurazione delle celle e la concentrazione chimica, utilizzando una mappa di colori appropriata (ad esempio, `COLORMAP_HOT` per il campo chimico). Le immagini vengono inoltre ridimensionate tramite `cv::resize()` per aumentare la risoluzione e migliorare la qualità.

5. **Creazione del layout video:**
   Viene definito un layout per l'output video che include sia la configurazione delle celle che la concentrazione chimica, posizionandoli fianco a fianco con titoli, legenda e coordinate esterne. Ogni immagine viene combinata con padding per creare un layout finale ordinato. Sono anche aggiunti contorni neri attorno ai dati per migliorarne la visibilità.

6. **Creazione della legenda:**
   Viene creata una barra dei colori verticale per rappresentare i valori del campo chimico. La funzione `createColorBar()` si occupa di creare un gradiente di colori, mentre `addColorBarLabels()` aggiunge le etichette numeriche per rappresentare la scala dei valori, rendendo così più chiara l'interpretazione dei dati.

7. **Aggiunta di titoli e testo:**
   Sopra ciascuna finestra di visualizzazione viene aggiunto un titolo per indicare cosa viene rappresentato: "*Cell Configuration*" per la disposizione delle celle e "*Chemical Concentration*" per il campo chimico. Inoltre, viene mostrato il numero del frame corrente per indicare l'avanzamento della simulazione.

8. **Salvataggio e visualizzazione del video:**
   Ogni frame generato viene scritto nel video e visualizzato sullo schermo con `imshow()` per la verifica in tempo reale della simulazione. Al termine del ciclo, il video viene rilasciato e tutte le finestre grafiche vengono chiuse.

La funzione `visualizeInit()` ha un ruolo fondamentale nella rappresentazione dei risultati della simulazione, fornendo una visualizzazione della configurazione cellulare e della distribuzione chimica.

2.3.3 Motivazioni per il passaggio dalla visualizzazione da MATLAB a OpenCV

Il passaggio dalla visualizzazione da MATLAB a OpenCV è stato motivato principalmente da esigenze di prestazioni e flessibilità. Nonostante l'intuitività di MATLAB, che risulta ideale per la prototipazione rapida e la visualizzazione dei dati, presenta delle limitazioni significative quando si tratta di gestire simulazioni con una grande quantità di dati. Le routine di visualizzazione in MATLAB, infatti, si sono dimostrate relativamente lente e dispendiose in termini di risorse, specialmente per simulazioni che richiedono la gestione di molti frame, come nel caso del Modello Cellulare di Potts.

Per questo motivo, si è scelto di sfruttare la programmazione C++ e la libreria OpenCV per migliorare la velocità di esecuzione e garantire una maggiore personalizzazione delle visualizzazioni. OpenCV offre un'ampia gamma di strumenti per l'elaborazione delle immagini, che consentono di gestire in modo efficiente operazioni come il ridimensionamento, l'applicazione di mappe di colori, l'aggiunta di testo e la sovrapposizione di elementi grafici. È chiaro che la versione in C++ richieda più lavoro nella scrittura del codice: ciò che può essere espresso in poche righe di codice MATLAB spesso necessita di molte più istruzioni in C++. Tuttavia, l'uso di OpenCV ha permesso di ottenere un controllo più granulare sugli aspetti estetici e funzionali delle immagini generate, portando a una visualizzazione più chiara e dinamica.

Un ulteriore vantaggio dell'integrazione di OpenCV nel progetto rispetto all'uso di MATLAB riguarda la sinergia con le altre componenti del sistema di simulazione, scritte in C e CUDA. Grazie a OpenCV, è stato possibile evitare il passaggio tra diversi ambienti di sviluppo, riducendo le dipendenze e aumentando la coerenza tra il codice di simulazione e quello di visualizzazione. Questo ha semplificato notevolmente il processo di sviluppo, minimizzando i costi computazionali e ottimizzando la pipeline di lavoro complessiva.

2.3.4 Confronto delle performance tra MATLAB e C

Per paragonare i risultati delle esecuzioni sequenziali del programma in MATLAB e in C, è stata impiegata una macchina con le seguenti specifiche:

- Processore: Intel(R) Core i7-6700HQ, 2.60 GHz, 4 Core
- RAM: 16 GB, DDR4-2133 MHz
- Disco: SSD 1TB

Per garantire un confronto equo, è stata utilizzata la stessa configurazione di simulazione per entrambi i programmi:

Tabella 2.1: Parametri di simulazione

Parametro	**Simulazione 1**	**Simulazione 2**	**Simulazione 3**
Dimensione del Reticolo	200x200	400x400	800x800
Numero di Cellule	10	400	80
Step di Monte Carlo	100	200	400
Temperatura	20	20	20

Tabella 2.2: Parametri del Modello Cellulare di Potts

Parametro	**Valore**
λV	50
λS	0.5
λC	100
λE	50
Volume Target	500

Tabella 2.3: Parametri del campo chimico

Parametro	**Valore**
D	0.1
Decay Rate	0.01
Secretion Rate	0.1
Diff DT	0.001

Per comprendere meglio l'entità dei dati relativi alla risoluzione del sistema lineare preso in analisi, si presenta una tabella riassuntiva che evidenzia la relazione diretta tra l'aumento delle dimensioni del reticolo e la crescita della dimensione della matrice A. Come si può osservare, l'incremento delle dimensioni del reticolo comporta una crescita significativa della matrice associata. Di conseguenza, l'approccio a matrice densa diventa impraticabile per dimensioni superiori a pochi elementi, rendendolo inadeguato per simulazioni scientifiche significative.

Tabella 2.4: Dimensioni del sistema lineare

Reticolo NxN	**Matrice A (complessiva)**	**Vettore b**
200x200	1600000000	40000
400x400	25600000000	160000
800x800	409600000000	640000

Tabella 2.5: Confronto di tempi di simulazione

	MATLAB	**C**
Simulazione 1	00:06:42.355	00:01:04.600
Simulazione 2	04:42:57.908	00:18:21.090
Simulazione 3	NC[1]	05:04:49.295

Tabella 2.6: Confronto di tempi di visualizzazione

	MATLAB	**C++ con OpenCV**
Simulazione 1	00:16:46	00:0:01
Simulazione 2	04:17:30	00:00:08
Simulazione 3	NC	00:01:01

Come evidenziato nella Tabella 2.5, l'implementazione in C ha portato a una riduzione significativa del tempo di esecuzione rispetto alla controparte MATLAB. Questo risultato è attribuibile principalmente all'uso di un linguaggio a basso livello, che consente una gestione più efficiente della memoria e un'ottimizzazione delle operazioni di calcolo.

Analizzando i risultati, per un reticolo di dimensioni 200x200, l'implementazione in C risulta circa 6 volte più veloce rispetto a MATLAB, utilizzando il tempo di esecuzione di MATLAB come riferimento. Per un reticolo più grande, di 400x400, il codice in C mostra un'accelerazione di circa 15 volte. Questo evidenzia chiaramente che l'implementazione in C offre una scalabilità superiore rispetto a MATLAB, rendendola ideale per simulazioni di dimensioni maggiori. Tuttavia, per il caso di un reticolo 800x800, non è stato possibile completare una simulazione completa. Il tempo richiesto per ogni singolo step, prima che l'esecuzione si interrompesse, è stato di circa 24 minuti. Ciò implica che per 400 Step di Monte Carlo il tempo stimato per l'intera simulazione sarebbe di circa 9600 minuti, ovvero 160 ore, corrispondenti a oltre 6 giorni. Poiché non è stato possibile portare a termine una simulazione completa, questi dati non sono stati inclusi nelle tabelle riassuntive.

È altrettanto interessante analizzare i tempi di visualizzazione riportati nella Tabella 2.6. I dati evidenziano che la visualizzazione con OpenCV risulta anch'essa scalabile, consentendo di completare l'operazione in pochi istanti, a differenza di MATLAB, dove i tempi richiesti sono significativamente maggiori.

Simulazioni a video

Di seguito vengono mostrati alcuni fotogrammi estratti dai video prodotti dalle simulazioni. Sebbene i parametri utilizzati siano identici per entrambe le versioni del programma, la natura stocastica delle componenti del modello comporta una variabilità nei risultati tra diverse esecuzioni.

[1]NC: Non classificato. La simulazione per un reticolo di dimensioni 800x800 non è stata completata a causa del tempo eccessivo richiesto per portarla a termine.

Questo comportamento è attribuibile alle scelte casuali intrinseche sia nell'algoritmo di Metropolis sia nel metodo Monte Carlo.

Come evidenziato dai tempi di esecuzione riassunti nella tabella 2.6, l'implementazione in C ha permesso un notevole miglioramento nei tempi di simulazione rispetto a quelli di MATLAB. Anche le routine di visualizzazione sviluppate in C++ con OpenCV hanno significativamente accelerato il processo di rendering, rendendo la visualizzazione estremamente veloce.

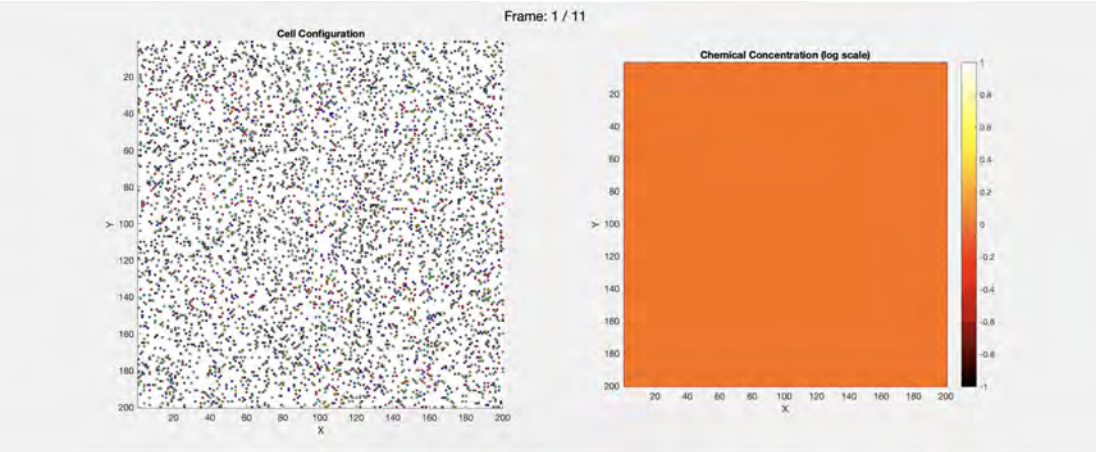

Figura 2.1: Frame 1 della visualizzazione in MATLAB, stato iniziale

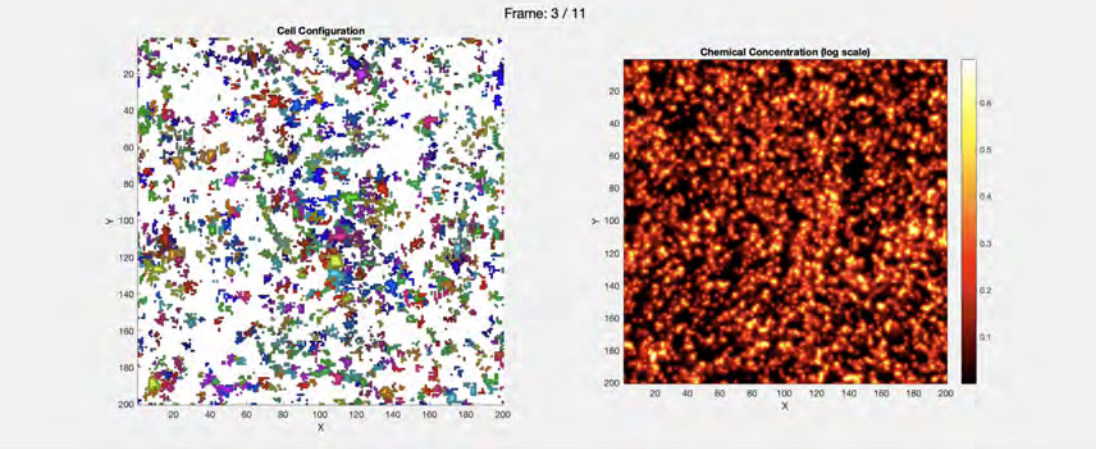

Figura 2.2: Frame 3 della visualizzazione in MATLAB

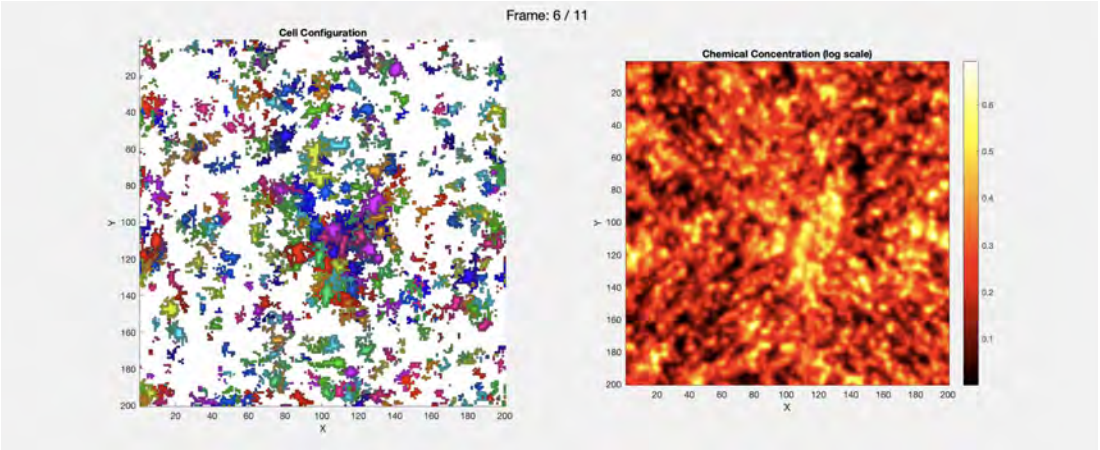

Figura 2.3: Frame 6 della visualizzazione in MATLAB

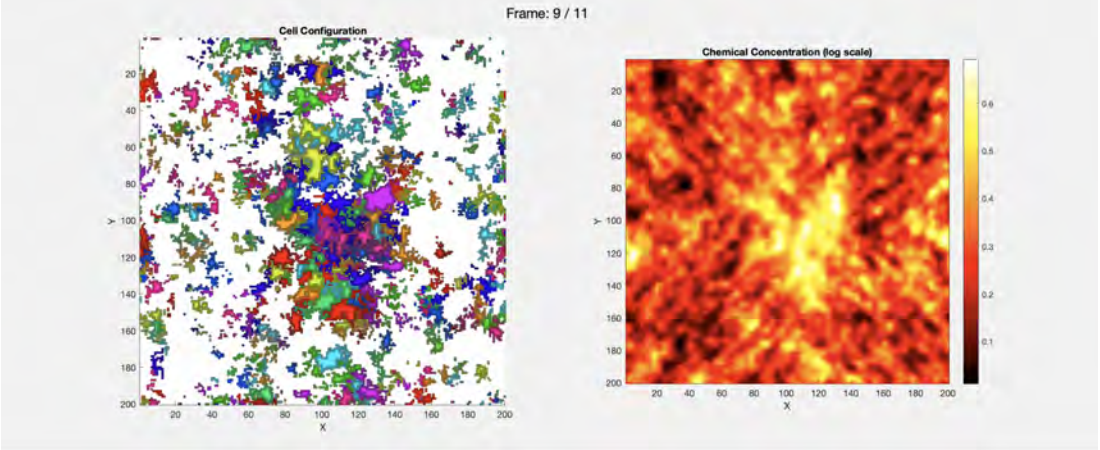

Figura 2.4: Frame 9 della visualizzazione in MATLAB

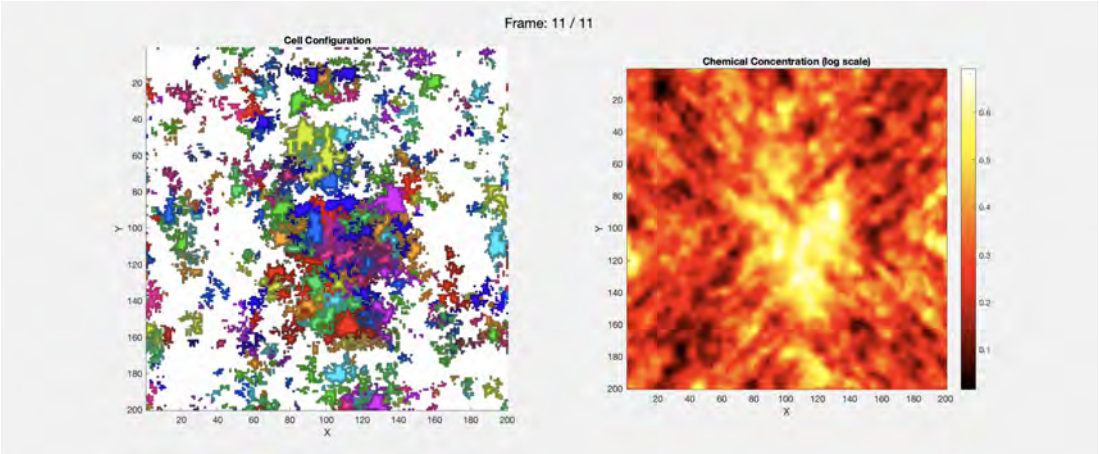

Figura 2.5: Frame 11 della visualizzazione in MATLAB

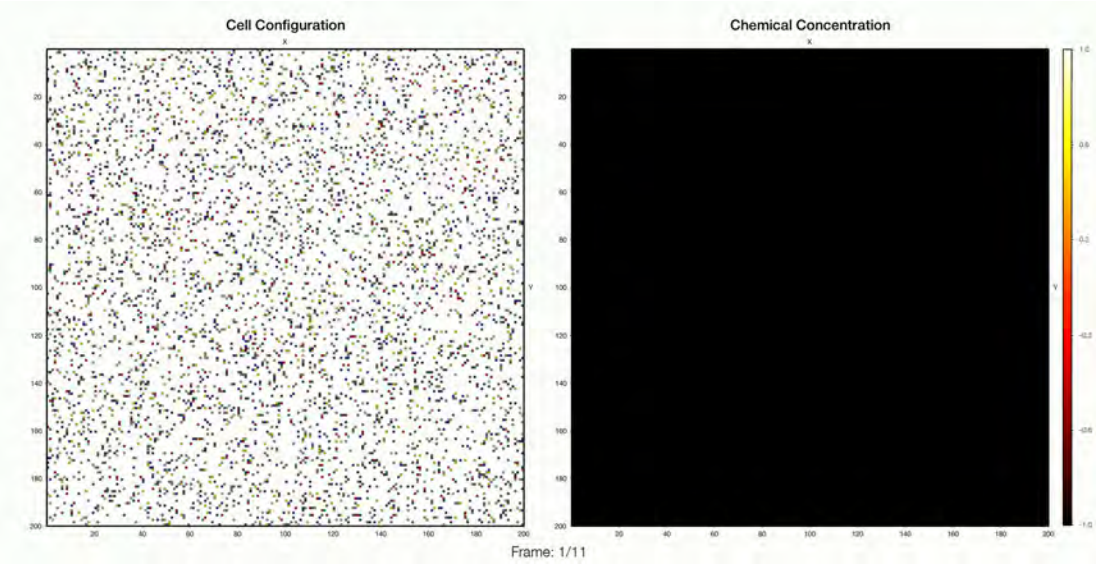

Figura 2.6: Frame 1 della visualizzazione in C++ con OpenCV, stato iniziale

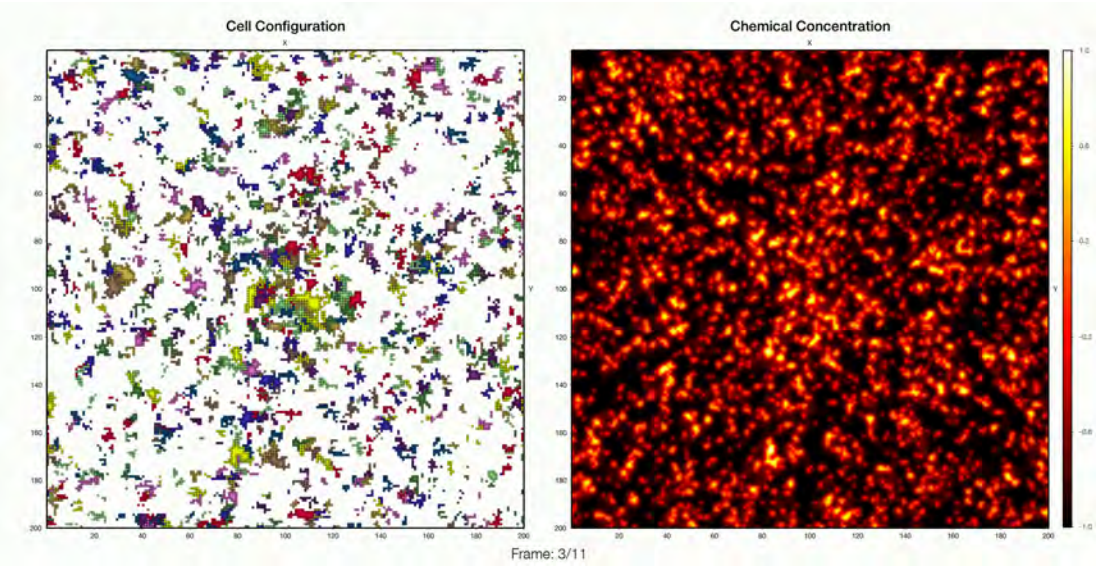

Figura 2.7: Frame 3 della visualizzazione in C++ con OpenCV

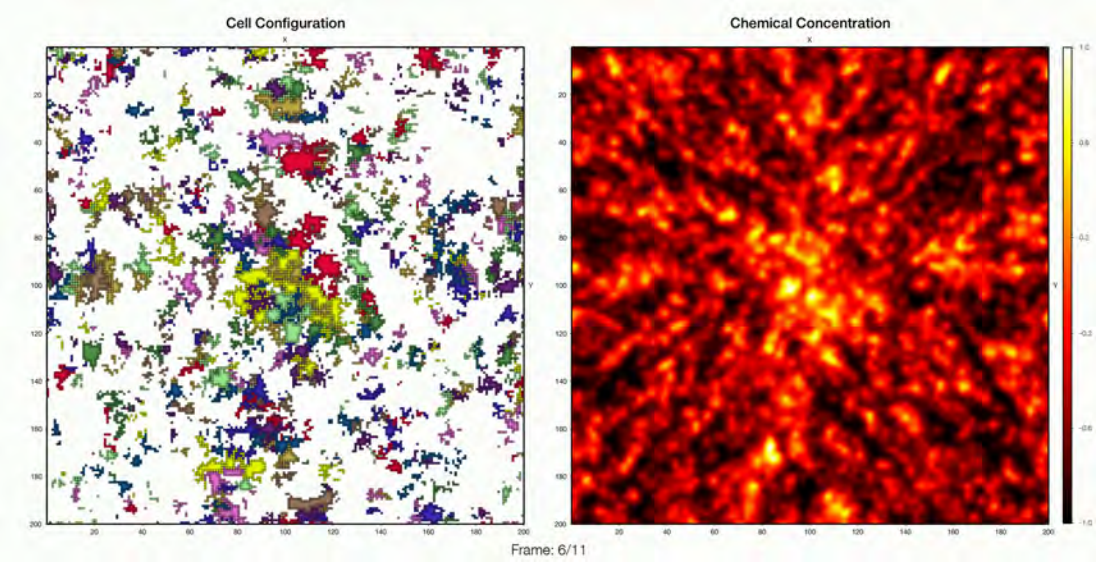

Figura 2.8: Frame 6 della visualizzazione in C++ con OpenCV

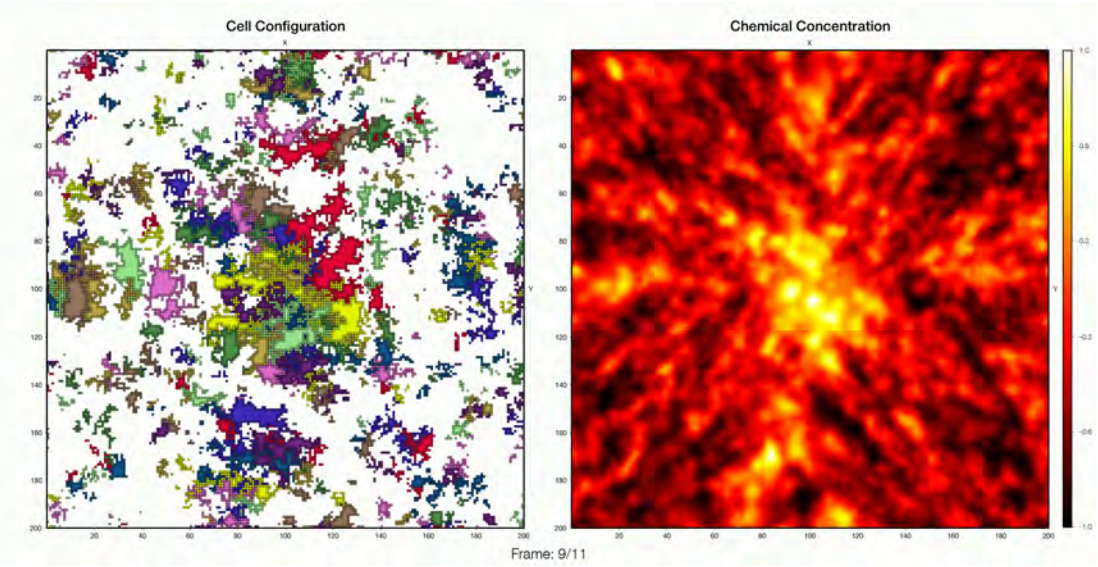

Figura 2.9: Frame 9 della visualizzazione in C++ con OpenCV

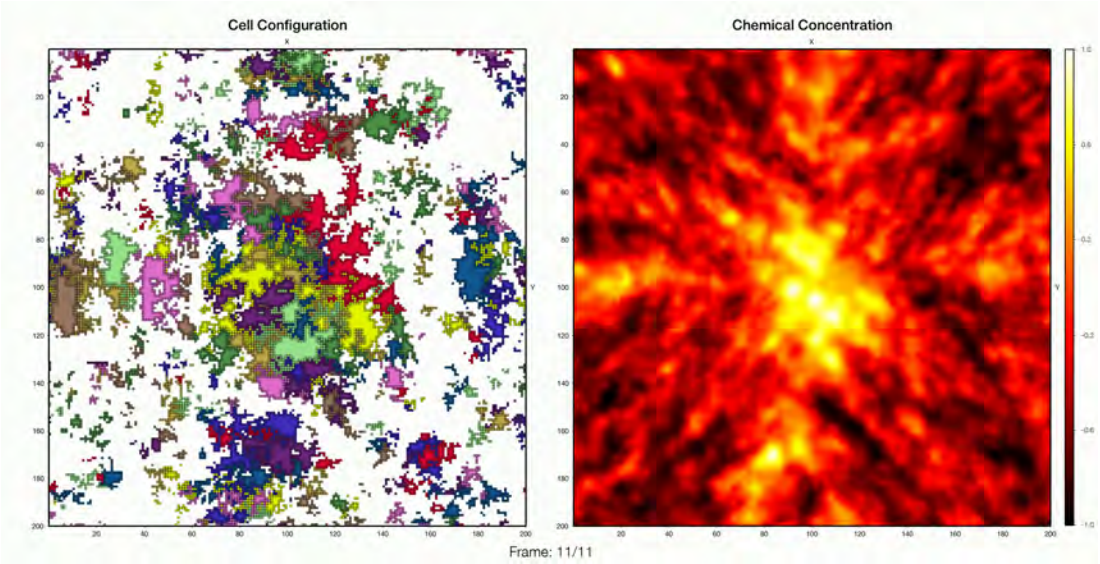

Figura 2.10: Frame 11 della visualizzazione in C++ con OpenCV

Capitolo 3

Architetture parallele e programmazione con CUDA

3.1 Introduzione alla computazione parallela e GPU

In questo capitolo verranno esplorati i concetti fondamentali della computazione parallela, con un particolare focus sui supercomputer, le architetture di sistemi paralleli, e le GPU (*Graphics Processing Units*). Il capitolo partirà da una descrizione generale della computazione ad alte prestazioni, passando poi ad approfondire i diversi tipi di parallelismo secondo la tassonomia di Flynn, per concludere con una dettagliata analisi delle GPU e della loro architettura ottimizzata per il parallelismo di tipo SIMD. Verrà inoltre presentata una panoramica delle differenze tra CPU e GPU e introdotto il concetto di GPGPU (*General-Purpose Computing on Graphics Processing Units*), che rappresenta l'uso delle GPU per compiti di calcolo generale oltre il *rendering* grafico. Infine, sarà fornita una descrizione dell'implementazione in parallelo tramite CUDA del codice CPM per la simulazione dell'angiogenesi.

3.1.1 Supercomputer e computazione ad alte prestazioni

I supercomputer rappresentano il vertice della potenza computazionale moderna, progettati per risolvere problemi estremamente complessi in tempi brevi [15] (*Jack Dongarra, 2019*). Diversamente dai computer tradizionali, essi sono costituiti da migliaia o milioni di unità di elaborazione che lavorano in parallelo, sfruttando architetture specializzate e una memoria distribuita per affrontare compiti intensi dal punto di vista computazionale. La computazione ad alte prestazioni (*High-Performance Computing*, HPC) si basa sul parallelismo per elaborare simultaneamente un numero enorme di operazioni e gestire grandi volumi di dati, rendendola ideale per applicazioni come simulazioni climatiche, modellazione molecolare, esplorazione spaziale e analisi di big data.

Le prestazioni di un supercomputer si misurano in FLOPS (*Floating Point Operations Per Second*) e i sistemi più avanzati possono superare il livello di petaflop, eseguendo un quadrilione di operazioni al secondo. Tali macchine sono costituite da migliaia di nodi di calcolo interconnessi tramite reti ad alta velocità, che garantiscono una comunicazione rapida tra i nodi, massimizzando l'efficienza e riducendo i tempi di calcolo. Grazie a queste caratteristiche, i supercomputer consentono di affronta-

re problemi impossibili da gestire con sistemi convenzionali, accelerando il progresso scientifico e tecnologico in diversi ambiti [42] (*TOP500 Organization*, *2024*).

Applicazioni dei supercomputer

I supercomputer trovano applicazione in molti campi della scienza e della tecnologia. Alcuni esempi di utilizzo includono:

- **Previsioni meteorologiche**: I supercomputer vengono utilizzati per creare modelli atmosferici molto dettagliati, prevedendo il tempo con un'elevata accuratezza. Possono analizzare miliardi di dati ambientali per determinare tendenze meteorologiche e anticipare eventi estremi come uragani o tempeste.
- **Simulazioni nucleari**: Permettono di simulare esplosioni nucleari senza la necessità di test fisici. Questo aiuta a garantire la sicurezza e la gestione degli arsenali nucleari, assicurando che siano mantenuti senza la necessità di effettuarne prove distruttive.
- **Ricerca medica e genomica**: Supercomputer sono impiegati per sequenziare il genoma umano e per simulare le interazioni tra farmaci e proteine. Questo tipo di calcolo è fondamentale per la scoperta di nuovi farmaci e per la comprensione delle malattie genetiche.
- **Astrofisica**: L'esplorazione dello spazio profondo e la modellazione del comportamento delle stelle e delle galassie richiedono simulazioni molto complesse che possono essere gestite solo con l'ausilio di supercomputer.
- **Ingegneria avanzata**: Nella progettazione di nuovi materiali e strutture, i supercomputer sono impiegati per simulare e analizzare il comportamento dei materiali in diverse condizioni, come carichi meccanici estremi o alte temperature.

Queste applicazioni beneficiano enormemente della capacità dei supercomputer di elaborare simultaneamente un grande numero di dati, riducendo significativamente i tempi di calcolo e rendendo possibili studi e sperimentazioni che richiederebbero troppo tempo o risulterebbero troppo costose se eseguite in maniera tradizionale.

È importante anche riflettere però sulla necessità di utilizzo di un supercomputer: sebbene i campi sopracitati richiedano usi intensi di risorse computazionali, un utente medio potrebbe non avere bisogno di una tale potenza di calcolo. La maggior parte delle applicazioni quotidiane, come l'elaborazione di testi, la navigazione web o anche la creazione di grafica per piccoli progetti, può essere gestita con computer di fascia media o addirittura dispositivi mobili. I supercomputer sono progettati per scenari altamente specializzati, dove il parallelismo e la gestione di enormi quantità di dati risultano fondamentali per raggiungere gli obiettivi desiderati.

Inoltre, l'uso di supercomputer comporta dei costi elevati sia in termini di infrastrutture che di consumo energetico. Di conseguenza, è necessario valutare attentamente il rapporto tra costi e benefici, e optare per il loro utilizzo solo quando le esigenze computazionali non possono essere soddisfatte con soluzioni più accessibili. Il progresso delle tecnologie GPU e dei sistemi di calcolo distribuito, tuttavia, ha reso possibile, negli ultimi anni, sfruttare una parte del potenziale dei supercomputer attraverso soluzioni più economiche e scalabili, anche per ricercatori e utenti con risorse limitate.

3.1.2 Tempi di esecuzione e efficienza computazionale

Il tempo di esecuzione e l'efficienza computazionale sono elementi fondamentali per valutare la qualità di un algoritmo e la sua implementazione [22] (*John L Hennessy e David A Patterson, 2011*). La riduzione del tempo di esecuzione e l'ottimizzazione dell'efficienza computazionale sono obiettivi cruciali, specialmente quando si lavora con grandi volumi di dati o quando la simulazione deve essere eseguita ripetutamente. Conoscere e analizzare il tempo di esecuzione permette di prendere decisioni informate su come migliorare un sistema, sia attraverso l'ottimizzazione del codice che tramite l'adozione di strategie di parallelizzazione. Inoltre, misurare l'efficienza computazionale è importante per garantire che le risorse di calcolo vengano utilizzate in modo efficace, minimizzando sprechi e riducendo i costi associati.

Tempo di esecuzione e performance

Il tempo di esecuzione di un algoritmo può essere rappresentato attraverso la seguente formula [50] (*Lloyd N Trefethen e David Bau III, 1997*):

$$\tau = K \cdot T(n) \cdot \mu$$

Dove:

- τ rappresenta il tempo totale di esecuzione dell'algoritmo.
- K è il fattore di scala, che rappresenta l'efficienza hardware e il numero di risorse a disposizione, come la velocità della CPU o il numero di unità di elaborazione parallele.
- $T(n)$ rappresenta la complessità computazionale, ossia il tempo teorico richiesto dall'algoritmo in funzione della dimensione del problema n. Ad esempio, un algoritmo con complessità $T(n) = O(n^2)$ richiederà più tempo di un algoritmo con complessità lineare quando la dimensione dei dati aumenta.
- μ rappresenta il tempo di lavoro per unità di operazione, che dipende dalle caratteristiche dell'hardware utilizzato.

Per ridurre il tempo di esecuzione τ, è possibile agire su diversi fattori:

- **Ottimizzare la complessità computazionale** ($T(n)$): scegliere o sviluppare algoritmi più efficienti può ridurre significativamente il tempo di esecuzione.
- **Aumentare il fattore di scala** (K): l'utilizzo di risorse computazionali aggiuntive, come CPU più potenti o GPU con più core, può ridurre il tempo richiesto.
- **Minimizzare il tempo per operazione** (μ): ciò può essere fatto migliorando l'efficienza delle singole operazioni, ad esempio sfruttando caratteristiche hardware specifiche, come il calcolo vettoriale su GPU o la pipeline di elaborazione delle CPU moderne.

Ottimizzare questi fattori richiede una comprensione profonda sia dell'algoritmo sia dell'hardware disponibile, così da ottenere la miglior combinazione possibile per ridurre τ al minimo.

Legge di Amdahl e scalabilità

La legge di Amdahl è un principio fondamentale per comprendere le limitazioni del parallelismo e l'efficienza potenziale di un sistema parallelo. La legge di Amdahl afferma che il guadagno ottenibile grazie alla parallelizzazione dipende dalla porzione del codice che può essere parallelizzata [1] (*Gene M. Amdahl, 1967*). Più precisamente, se una frazione P di un programma è parallelizzabile, mentre la parte rimanente $(1-P)$ è intrinsecamente seriale, il miglioramento massimo della velocità, con N processori, è dato dalla seguente formula:

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

Dove:

- $S(N)$ è il fattore di accelerazione.
- P è la frazione del programma parallelizzabile.
- N è il numero di processori utilizzati.

Questo significa che anche con un numero infinito di processori, il miglioramento sarà sempre limitato dalla porzione seriale del programma. Ad esempio, se solo il 90% del programma è parallelizzabile, il miglioramento massimo possibile sarà di circa 10 volte, indipendentemente dal numero di processori.

La scalabilità è strettamente legata alla legge di Amdahl e rappresenta la capacità di un sistema di aumentare le sue prestazioni all'aumentare delle risorse disponibili. Quando un sistema è scalabile, il tempo di esecuzione diminuisce significativamente all'aumentare delle risorse computazionali. Tuttavia, la presenza di parti non parallelizzabili limita la scalabilità, ed è per questo che un buon design dell'algoritmo è fondamentale per massimizzare il parallelismo.

La legge di Amdahl aiuta a comprendere i limiti teorici dell'accelerazione attraverso il parallelismo, e questo è cruciale per valutare la scalabilità delle soluzioni proposte e per determinare se l'incremento del numero di risorse porterà effettivamente a una riduzione dei tempi di esecuzione. Essa fornisce anche indicazioni utili per concentrare gli sforzi sull'ottimizzazione delle parti seriali, che spesso rappresentano il collo di bottiglia più significativo in un sistema altamente parallelo.

Speedup: misurazione delle prestazioni

Lo *speedup* è una metrica essenziale per valutare il miglioramento delle prestazioni di un'implementazione ottimizzata rispetto a una di riferimento [14] (*James W Demmel, 1997*). Esso è definito come il rapporto tra il tempo di esecuzione della versione originale di un algoritmo e il tempo di esecuzione della versione ottimizzata:

$$S = \frac{T_{\text{originale}}}{T_{\text{ottimizzato}}}$$

Dove:

- $T_{originale}$ rappresenta il tempo di esecuzione della versione originale.
- $T_{ottimizzato}$ rappresenta il tempo di esecuzione della versione ottimizzata.

Uno speedup maggiore di 1 indica un miglioramento delle prestazioni, con valori più alti che rappresentano guadagni più significativi.

L'importanza dello speedup risiede nella sua capacità di fornire una misura quantitativa del guadagno in efficienza. Nel contesto del calcolo ad alte prestazioni, esso è particolarmente utile per valutare l'efficacia della parallelizzazione e giustificare i costi aggiuntivi associati all'uso di risorse computazionali avanzate, come GPU o cluster.

Va comunque notato che lo speedup non tiene conto direttamente dell'overhead introdotto dall'ottimizzazione, come il costo aggiuntivo della gestione delle risorse parallele o il tempo richiesto per trasferimenti di dati tra CPU e GPU. Inoltre, in scenari di parallelizzazione, lo speedup può essere influenzato dalla presenza di parti non parallelizzabili, come descritto dalla legge di Amdahl.

3.1.3 Tipi di parallelismo

Il parallelismo è una tecnica che consente di eseguire più operazioni simultaneamente, migliorando l'efficienza e riducendo i tempi di calcolo di un sistema. Esistono diversi tipi di parallelismo, ciascuno con caratteristiche e applicazioni specifiche. Capire i vari tipi di parallelismo è fondamentale per scegliere l'approccio più adatto a risolvere problemi di diversa natura, sia dal punto di vista dell'architettura hardware sia dell'implementazione software. In questa sezione, verranno illustrati i principali tipi di parallelismo, suddivisi in base alla tassonomia di Flynn.

Tassonomia di Flynn

La tassonomia di Flynn è una classificazione dei modelli di architettura computazionale da Michael J. Flynn nel 1972 [17] (*Michael J Flynn, 1972*). Basata sulla natura delle istruzioni e dei dati che vengono processati simultaneamente. Questa tassonomia è utile per comprendere i diversi approcci alla parallelizzazione e per definire le caratteristiche principali dei sistemi di calcolo. La tassonomia di Flynn distingue quattro categorie principali: SISD, SIMD, MISD e MIMD. In questa sezione verranno descritti in dettaglio i diversi tipi, con un focus sulle loro peculiarità e applicazioni principali.

SISD

Il modello SISD (*Single Instruction, Single Data*) rappresenta l'approccio tradizionale alla computazione sequenziale. In un sistema SISD, un singolo processore esegue una sola istruzione su un singolo flusso di dati alla volta. Questo è il tipo di architettura più semplice, comunemente utilizzato nei primi calcolatori e tuttora presente nei processori per uso generale. Un sistema SISD non prevede alcuna forma di parallelismo e rappresenta il punto di partenza per comprendere le limitazioni della computazione seriale.

SIMD

Il modello SIMD (*Single Instruction, Multiple Data*) permette di eseguire la stessa istruzione simultaneamente su più dati. Questo tipo di parallelismo è particolarmente utile per problemi che richiedono l'applicazione dello stesso operatore su grandi

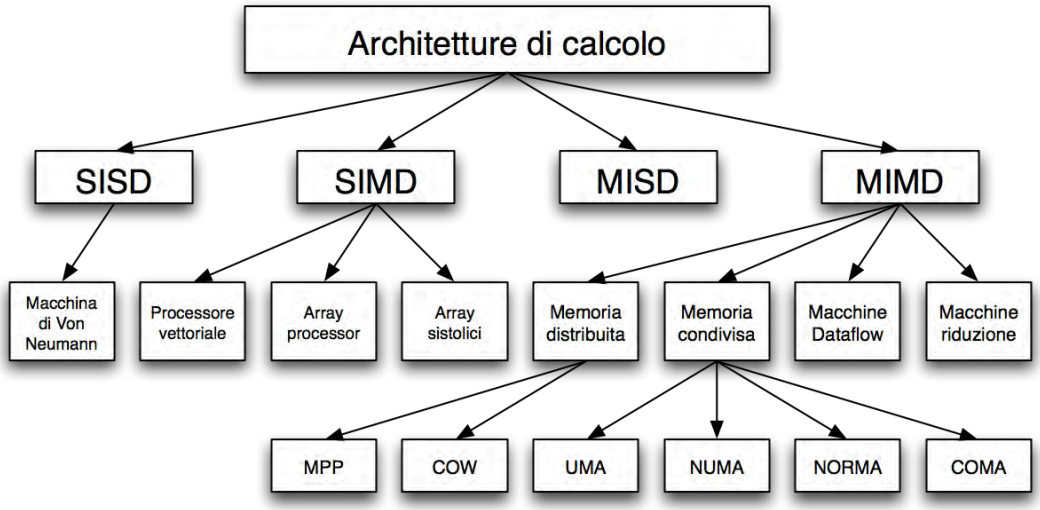

Figura 3.1: Tassonomia di Flynn

volumi di dati, come nei calcoli vettoriali e nelle operazioni grafiche. Le GPU (*Graphics Processing Units*) utilizzano un'architettura basata su SIMD per eseguire operazioni di elaborazione delle immagini, dove molteplici pixel devono essere manipolati contemporaneamente in modo simile.

MISD

Il modello MISD (*Multiple Instruction, Single Data*) prevede l'applicazione di istruzioni diverse allo stesso flusso di dati. Questo tipo di parallelismo è raro e trova applicazioni limitate, spesso in sistemi con esigenze di tolleranza ai guasti o dove è necessario applicare più controlli o elaborazioni in parallelo su un unico dato. La complessità e la limitata utilità di questo approccio ne rendono l'uso molto meno diffuso rispetto alle altre architetture.

MIMD

Il modello MIMD (*Multiple Instruction, Multiple Data*) è il più flessibile e potente tra quelli descritti nella tassonomia di Flynn. In un sistema MIMD, più processori eseguono istruzioni diverse su dati differenti, permettendo una notevole capacità di elaborazione parallela. Questo tipo di architettura è utilizzato nei supercomputer e nei sistemi di elaborazione distribuita, dove ogni processore può essere dedicato a compiti diversi, garantendo così una notevole capacità di parallelismo per applicazioni complesse come la simulazione scientifica e l'elaborazione di dati massivi.

Sistemi a memoria condivisa

Nei sistemi a memoria condivisa, tutti i processori hanno accesso alla stessa memoria centrale. Questo tipo di architettura è utile per facilitare la comunicazione tra i processori, poiché condividono lo stesso spazio di memoria e possono accedere direttamente ai dati degli altri processori. Tuttavia, ciò può portare a colli di bottiglia quando molti processori accedono contemporaneamente alla memoria, riducendo così

l'efficienza del sistema. I sistemi a memoria condivisa sono spesso utilizzati in server multiprocessore e workstation ad alte prestazioni.

Sistemi a memoria distribuita

Nei sistemi a memoria distribuita, ogni processore ha una memoria locale e non può accedere direttamente alla memoria degli altri processori. Invece, la comunicazione avviene attraverso la rete, che connette i vari nodi del sistema. Questo tipo di architettura è altamente scalabile e adatta a grandi sistemi di calcolo, come i supercomputer, poiché l'accesso alla memoria è decentralizzato, eliminando così i colli di bottiglia tipici dei sistemi a memoria condivisa. Tuttavia, la necessità di comunicare attraverso la rete introduce una latenza che deve essere gestita con opportune strategie per ottimizzare il parallelismo.

3.1.4 GPU e il modello SIMD

Le GPU (*Graphics Processing Units*) sono progettate per elaborare un'enorme quantità di operazioni in parallelo, rendendole estremamente efficaci per problemi che richiedono l'elaborazione simultanea di grandi volumi di dati. Originariamente sviluppate per accelerare il *rendering* grafico, le GPU si sono evolute per supportare applicazioni computazionali generali grazie all'adozione del modello SIMD (*Single Instruction, Multiple Data*). Questo modello consente l'esecuzione della stessa istruzione su molteplici unità di dati in parallelo, sfruttando il parallelismo a livello di dati per migliorare significativamente le prestazioni rispetto all'elaborazione sequenziale delle CPU.

Il modello SIMD è particolarmente vantaggioso in applicazioni dove operazioni identiche devono essere eseguite su grandi insiemi di dati, come nell'elaborazione delle immagini, nell'analisi dei dati e nelle simulazioni scientifiche. Ad esempio, nel processamento di immagini, il modello SIMD permette di applicare filtri o trasformazioni a migliaia di pixel contemporaneamente, sfruttando l'ampia capacità di parallelismo delle GPU.

Architettura GPU

Le GPU sono progettate per massimizzare il *throughput*, ovvero il numero di operazioni eseguite per unità di tempo, piuttosto che minimizzare la latenza delle singole operazioni come avviene nelle CPU [35] (*NVIDIA*, *2020*). Una GPU è composta da un gran numero di core o unità di elaborazione, organizzati in *Streaming Multiprocessor* (SM). Ogni SM contiene molteplici core che lavorano insieme per eseguire gruppi di thread in parallelo.

Un elemento chiave dell'architettura GPU è l'uso dei *warp*, gruppi di thread (tipicamente 32) che eseguono la stessa istruzione simultaneamente su dati diversi [23] (*John L Hennessy e David A Patterson*, *2017*). Questo approccio, in linea con il modello SIMD, massimizza sia l'efficienza dell'hardware che il throughput. Le GPU dispongono inoltre di una memoria gerarchica che include memoria globale, memoria condivisa e registri. La memoria globale è accessibile a tutti i thread, mentre la memoria condivisa permette una comunicazione rapida tra thread appartenenti allo stesso blocco, migliorando le prestazioni in applicazioni che richiedono scambio di dati tra thread [28] (*David B Kirk e Wen-Mei W Hwu*, *2017*).

Questo design rende le GPU estremamente efficienti per problemi computazionali intensivi che possono sfruttare il parallelismo massivo, come le simulazioni scientifiche, la grafica tridimensionale e il machine learning. La capacità di eseguire migliaia di thread simultaneamente offre un vantaggio significativo rispetto alle CPU, che sono ottimizzate per gestire un numero limitato di thread con bassa latenza.

3.1.5 Confronto tra CPU e GPU

Le CPU (*Central Processing Unit*) e le GPU (*Graphics Processing Units*) sono due tipi distinti di processori con architetture e finalità progettuali differenti. La CPU è l'unità centrale di elaborazione dati di un computer, progettata principalmente per eseguire un numero limitato di thread in modo altamente efficiente, gestendo operazioni sequenziali e di controllo. Al contrario, la GPU è stata sviluppata per accelerare il rendering grafico e gestire grandi quantità di dati in parallelo grazie a un'architettura altamente parallelizzata, capace di eseguire migliaia di thread contemporaneamente.

Le CPU hanno un numero relativamente basso di core rispetto alle GPU, ma ciascun core è estremamente potente e progettato per una vasta gamma di operazioni. Questo le rende ideali per attività che richiedono flessibilità e controllo su flussi di esecuzione complessi. Tipicamente, le CPU vengono utilizzate per gestire compiti che necessitano di decisioni sequenziali, come l'esecuzione di software applicativo, il coordinamento delle risorse del sistema operativo e la gestione dell'interazione con l'utente. Nel contesto della parallelizzazione con le CPU, secondo la tassonomia di Flynn, il modello di parallelismo più indicato è quello MIMD, grazie alla capacità dei core delle CPU di eseguire elaborazioni indipendenti e relativamente complesse.

Le GPU, invece, sono dotate di un numero elevato di core, ognuno dei quali esegue operazioni semplici su grandi volumi di dati. Questa caratteristica rende le GPU particolarmente adatte al parallelismo a livello di dati, ad esempio per il rendering di immagini, simulazioni fisiche, apprendimento automatico e, nel caso specifico trattato in questa tesi, la simulazione di modelli biologici. L'architettura delle GPU consente un uso intensivo del modello SIMD in cui la stessa istruzione viene applicata simultaneamente a grandi insiemi di dati, garantendo un elevato throughput computazionale.

Da ciò che è stato appena descritto, è evidente che le CPU e le GPU sono complementari. La prima è ideale per affrontare problemi legati al controllo del flusso di elaborazione, mentre la seconda è particolarmente adatta per il trattamento parallelo di grandi volumi di dati. La scelta tra CPU e GPU dipende strettamente dalla natura del problema da risolvere: la CPU è perfetta per compiti che richiedono procedure decisionali lineari e rapide, mentre il parallelismo massiccio, come nel rendering grafico o nelle simulazioni scientifiche, implica chiaramente l'uso della GPU.

Tabella 3.1: Confronto tra CPU e GPU

Caratteristica	**CPU**	**GPU**
Numero di Core	Pochi (fino a decine)	Molti (centinaia o migliaia)
Tipologia di Core	Potenti e versatili	Semplici e numerosi
Ottimizzazione	Sequenziale	Parallela
Modello di Parallelismo	MIMD	SIMD
Prestazioni	Operazioni di controllo e flussi complessi	Elaborazione massiva e ripetitiva
Applicazioni Tipiche	Sistemi operativi, software generici	Rendering grafico, machine learning, simulazioni
Capacità di Gestione dei Thread	Limitata, ottimizzata per pochi thread	Elevata, ottimizzata per migliaia di thread

3.1.6 GPGPU (General-Purpose Computing on Graphics Processing Units)

GPGPU, acronimo di *General-Purpose Computing on Graphics Processing Units*, si riferisce all'utilizzo delle GPU per scopi generici che vanno oltre la grafica e il rendering, sfruttando la potenza parallela delle GPU per elaborare operazioni computazionali intense. In passato, le GPU erano limitate esclusivamente a compiti di visualizzazione grafica, come la gestione di scene tridimensionali o il rendering di immagini. Tuttavia, con l'evoluzione dell'architettura delle GPU e dei linguaggi di programmazione, è diventato possibile utilizzare queste unità per una vasta gamma di applicazioni.

La GPGPU permette di sfruttare la struttura estremamente parallela delle GPU per eseguire calcoli di tipo scientifico e ingegneristico, simulazioni fisiche, *machine learning*, elaborazione di segnali e dati, e molte altre operazioni che beneficiano dell'esecuzione simultanea su un grande numero di thread. Grazie alla capacità delle GPU di eseguire lo stesso tipo di operazione su molti dati contemporaneamente, la GPGPU rappresenta una soluzione ideale per i problemi che possono essere parallelizzati in maniera efficiente, sfruttando il modello di calcolo SIMD (*Single Instruction, Multiple Data*).

L'utilizzo della GPGPU ha rivoluzionato molti settori, tra cui quello delle simulazioni scientifiche e del machine learning. Ad esempio, algoritmi di *deep learning* sono stati notevolmente accelerati grazie alla possibilità di utilizzare GPU per l'addestramento di reti neurali, che richiede l'esecuzione di numerose operazioni matriciali in parallelo.

L'introduzione di librerie e framework come CUDA di NVIDIA e OpenCL ha reso possibile e accessibile la programmazione delle GPU per scopi generici. Questi strumenti offrono agli sviluppatori la possibilità di scrivere codice che sfrutti direttamente la capacità di calcolo massivo delle GPU, ottimizzando le prestazioni e riducendo drasticamente i tempi di elaborazione rispetto all'esecuzione su CPU tradizionali.

Tuttavia, programmare per GPU richiede un'attenzione particolare nella gestione del parallelismo, nella sincronizzazione dei thread e nell'accesso efficiente alla memoria, poiché la natura delle GPU si presta principalmente ad operazioni di tipo *data parallel*.

3.1.7 Introduzione a CUDA

CUDA (*Compute Unified Device Architecture*) è una piattaforma di calcolo parallelo e un modello di programmazione sviluppato da NVIDIA per sfruttare la potenza delle GPU nell'elaborazione generale, oltre alla grafica. Fornisce strumenti e funzioni basate su un linguaggio di programmazione simile al C, rendendo accessibile lo sviluppo parallelo a un'ampia comunità di programmatori.

Un elemento chiave di CUDA è la distinzione tra *host* (CPU) e *device* (GPU). La CPU coordina l'intero programma, prendendo decisioni e delegando compiti alla GPU, che esegue operazioni in parallelo grazie alla sua architettura a migliaia di core. Questo modello collaborativo ottimizza i carichi di lavoro computazionali intensivi, come l'elaborazione di grandi quantità di dati.

In CUDA, le funzioni *kernel* definiscono le operazioni che la GPU esegue in parallelo. Questi kernel vengono configurati specificando il numero di thread e blocchi, che determina l'organizzazione del calcolo. L'esecuzione segue il modello SIMT (*Single Instruction Multiple Threads*), in cui gruppi di thread eseguono istruzioni condivise, mantenendo comunque un certo grado di indipendenza.

3.1.8 Concetti fondamentali

Host e device

CUDA distingue nettamente tra host (CPU) e device (GPU). La CPU gestisce la pianificazione e il coordinamento delle operazioni, mentre la GPU si occupa dell'esecuzione di calcoli intensivi sfruttando i suoi numerosi core. Questa separazione consente una divisione efficiente dei ruoli: la CPU gestisce le decisioni di alto livello e i flussi di controllo, delegando alla GPU compiti computazionali ripetitivi e parallelizzabili.

Un aspetto cruciale del modello CUDA è il trasferimento esplicito dei dati tra memoria dell'host e memoria del device. Prima di eseguire operazioni sulla GPU, i dati devono essere copiati nella sua memoria; successivamente, i risultati vengono trasferiti nuovamente alla memoria della CPU. Questo passaggio richiede attenzione per ottimizzare le prestazioni complessive dell'applicazione.

Funzioni kernel

Le funzioni kernel sono il cuore dell'elaborazione in CUDA, progettate per essere eseguite in parallelo dalla GPU. Ogni kernel è composto da molteplici thread, ciascuno incaricato di un'operazione su un sottoinsieme di dati. Ad esempio, in un calcolo su array di dimensione N, ogni thread può elaborare un elemento specifico, riducendo drasticamente il tempo richiesto rispetto all'elaborazione sequenziale.

I kernel vengono lanciati organizzando i thread in blocchi, e i blocchi in una griglia. Questa configurazione permette di distribuire il lavoro in maniera granulare e scalabile. All'interno di ogni blocco, i thread possono condividere memoria locale e sincronizzarsi per eseguire operazioni coordinate, un aspetto fondamentale per ottimizzare l'elaborazione parallela.

SIMT

L'esecuzione dei kernel su GPU segue il modello SIMT (*Single Instruction Multiple Threads*), una variante del modello SIMD che consente a gruppi di thread di eseguire la stessa istruzione contemporaneamente, con una particolare attenzione alla loro indipendenza. In CUDA, i thread sono raggruppati in unità chiamate warp. Ogni warp esegue le stesse istruzioni simultaneamente su dati diversi, ma i singoli thread all'interno del warp possono divergere in caso di istruzioni condizionali, seppur con un costo in termini di efficienza. Questo modello si adatta perfettamente alle GPU, che sono progettate per massimizzare il throughput su compiti paralleli e indipendenti.

L'organizzazione dei thread in griglie e blocchi e l'esecuzione in modalità SIMT consentono di sfruttare appieno il potenziale di calcolo parallelo della GPU, specialmente per operazioni su grandi dataset dove il lavoro può essere suddiviso facilmente in parti indipendenti.

3.1.9 Organizzazione dei thread in CUDA

Blocchi e griglie

In CUDA, l'organizzazione dei thread si basa su una struttura gerarchica di blocchi e griglie, che consente di suddividere l'elaborazione di grandi dataset e scalare l'applicazione su un numero elevato di core della GPU.

Un blocco è costituito da un gruppo di thread che eseguono la stessa funzione kernel su dati diversi. I thread di un blocco possono comunicare tra loro utilizzando la memoria condivisa, una cache locale ad alta velocità, che li supporta nella gestione di calcoli più complessi e nella condivisione di risultati intermedi.

I blocchi sono organizzati in una griglia, che rappresenta l'intero insieme di thread destinati a eseguire il kernel. La configurazione della griglia e dei blocchi, definita tramite la sintassi `<<<griglia, blocco>>>`, può essere adattata alle dimensioni del dataset e alle specifiche hardware della GPU, garantendo un utilizzo ottimale delle risorse e del parallelismo disponibile.

Gerarchia dei thread

In CUDA, la gerarchia dei thread comprende tre livelli principali, ciascuno con un ruolo specifico nell'elaborazione parallela:

- **Thread**: Un thread rappresenta l'unità base di esecuzione all'interno della GPU. Ogni thread ha il proprio indice unico che viene utilizzato per identificare il pezzo di dati su cui deve lavorare.
- **Blocco di thread**: Un blocco è un gruppo di thread che può cooperare tra loro tramite memoria condivisa e sincronizzazioni. I thread all'interno di un blocco sono limitati in numero, determinato dalla capacità hardware della GPU.
- **Griglia di blocchi**: La griglia rappresenta l'insieme dei blocchi lanciati per eseguire un kernel. I blocchi di una griglia sono indipendenti tra loro e possono essere eseguiti in parallelo su unità diverse della GPU.

Questa struttura gerarchica permette di mappare con precisione il lavoro parallelo sui dati, ottimizzando il bilanciamento del carico e l'efficienza computazionale.

Sincronizzazione dei thread

La sincronizzazione dei thread è un aspetto cruciale nell'elaborazione parallela per garantire la coerenza dei dati e il corretto ordine delle operazioni. In CUDA, la sincronizzazione può essere gestita a diversi livelli:

- **All'interno di un blocco**:

  I thread di un blocco possono essere sincronizzati tramite una chiamata alla funzione `__syncthreads()`. Questa funzione assicura che tutti i thread del blocco raggiungano un punto comune di esecuzione prima di proseguire, garantendo che i dati nella memoria condivisa siano completamente aggiornati.

- **Tra blocchi e l'host**:

  Per sincronizzare tutti i blocchi e garantire che il dispositivo (GPU) abbia completato l'esecuzione di un kernel prima di proseguire con altre operazioni sull'host (CPU), si usa la funzione `cudaDeviceSynchronize()`. Questa funzione forza il programma a bloccare l'esecuzione dell'host finché tutte le operazioni in corso sulla GPU non sono terminate. È particolarmente utile per verificare eventuali errori durante l'esecuzione del kernel o per assicurarsi che i dati prodotti dalla GPU siano pronti per essere trasferiti o utilizzati.

Mentre `__syncthreads()` è limitata ai thread all'interno di un singolo blocco, `cudaDeviceSynchronize()` opera a livello globale tra host e device, fornendo un controllo più ampio sul flusso di esecuzione. Tuttavia, usare in modo eccessivo la funzione `cudaDeviceSynchronize()` può introdurre overhead e ridurre le prestazioni, pertanto è consigliabile utilizzarla solo quando necessario, ad esempio per debugging o sincronizzazioni cruciali.

Esecuzione in warp

I thread in CUDA sono organizzati in unità chiamate warp, composte tipicamente da 32 thread. I warp rappresentano l'unità di esecuzione effettiva sulla GPU: tutti i thread di un warp eseguono la stessa istruzione contemporaneamente. Tuttavia, se i thread all'interno di un warp devono seguire percorsi condizionali differenti (divergenza del warp), ciò può rallentare significativamente l'esecuzione, poiché il warp deve completare ciascun percorso sequenzialmente.

Un altro fattore critico per l'efficienza dell'esecuzione in warp è la *coalescenza* della memoria. Questo concetto si riferisce alla modalità con cui i thread accedono alla memoria globale della GPU. Per massimizzare il throughput della memoria, i thread di un warp devono accedere a indirizzi di memoria globali consecutivi, consentendo l'unione di più richieste in una singola operazione di memoria. Se i thread accedono a indirizzi non allineati o non consecutivi, si verificano operazioni di memoria non coalescenti, che riducono drasticamente le prestazioni a causa di un maggiore numero di transazioni di memoria richieste.

La conoscenza dell'esecuzione in warp e dei principi di coalescenza è fondamentale per ottimizzare il codice CUDA. La progettazione di algoritmi dovrebbe minimizzare sia la divergenza dei warp che gli accessi non coalescenti, garantendo un utilizzo efficiente delle risorse della GPU e migliorando il parallelismo effettivo.

3.1.10 Struttura di un'applicazione CUDA

La struttura di un'applicazione CUDA segue un modello in cui l'host ha il ruolo principale di controllare il flusso dell'esecuzione e delegare i compiti di calcolo massivamente parallelo al device [38] (*NVIDIA Corporation*, *2024*). Questo paradigma si basa sulla collaborazione tra host e device per massimizzare l'efficienza e sfruttare al meglio le risorse di calcolo. Un'applicazione CUDA si articola generalmente nelle seguenti fasi:

Inizializzazione e allocazione della memoria

La prima fase dell'esecuzione di un'applicazione CUDA prevede l'inizializzazione dell'ambiente CUDA e l'allocazione della memoria necessaria per gestire i dati. In particolare, l'host si occupa di allocare memoria sia sull'host stesso sia sul device, assicurandosi che i dati siano trasferiti correttamente tra la CPU e la GPU, poiché queste non condividono una memoria comune.

Allocazione della memoria sull'host

Per allocare memoria sulla CPU (host), si utilizza la funzione standard `malloc()` in C, che permette di riservare blocchi di memoria di dimensioni specificate. Per esempio:
`int *hostArray = (int *)malloc(size * sizeof(int));`
In alternativa, `calloc()` può essere utilizzata per allocare memoria inizializzata a zero, utile quando i dati devono essere azzerati prima dell'elaborazione:
`int *hostArray = (int *)calloc(size, sizeof(int));`
Tuttavia, l'inizializzazione a zero con `calloc()` è più lenta rispetto all'uso combinato di `malloc()` e una successiva funzione come `memset()` per impostare i valori a zero, in base alle esigenze.

3.1.11 Allocazione della memoria sul device

Per allocare memoria sulla GPU (device), CUDA fornisce la funzione `cudaMalloc()`, che riserva uno spazio nella memoria globale del device. Ad esempio:

```
int *deviceArray;
cudaMalloc((void **)&deviceArray, size * sizeof(int));
```

Analogamente a `calloc()` sull'host, CUDA permette di inizializzare la memoria a zero utilizzando `cudaMemset()`. Questa funzione è utile per preparare la memoria prima dell'elaborazione:
`cudaMemset(deviceArray, 0, size * sizeof(int));`
È importante notare che, a differenza di `calloc()`, `cudaMemset()` non effettua l'allocazione ma semplicemente imposta a zero i valori di una memoria già allocata.

Preferenza per strutture monodimensionali

Per ottimizzare l'efficienza e ridurre il rischio di accessi non coalescenti alla memoria, è comune allocare strutture dati monodimensionali sia sull'host che sul device. Questo approccio facilita la coalescenza della memoria, che avviene quando i thread accedono a indirizzi consecutivi nella memoria globale del device. L'uso di strutture

bidimensionali o più complesse spesso comporta un accesso non contiguo alla memoria, riducendo il throughput. Ad esempio, per allocare una matrice bidimensionale come array monodimensionale:

```
int *deviceMatrix;
cudaMalloc((void **)&deviceMatrix, rows * cols * sizeof(int));
```

L'accesso agli elementi avverrà poi tramite un calcolo dell'indice:

```
int index = row * cols + col;
deviceMatrix[index] = value;
```

Questo approccio, combinato con una progettazione attenta degli algoritmi, permette di massimizzare le prestazioni delle applicazioni CUDA, sfruttando appieno la coalescenza della memoria e il parallelismo della GPU.

Copia dei dati tra host e device

Una volta allocata la memoria, i dati necessari vengono copiati dalla memoria dell'host alla memoria del device. Per eseguire questo trasferimento, CUDA fornisce la funzione `cudaMemcpy()`, che richiede di specificare la direzione del trasferimento tramite il parametro `cudaMemcpyKind`. Ad esempio:

```
cudaMemcpy(devA, hostA, size * sizeof(int), cudaMemcpyHostToDevice);
```

In questo caso, `cudaMemcpyHostToDevice` indica che i dati devono essere trasferiti dall'host al device. È è possibile utilizzare `cudaMemcpyAsync()` per trasferimenti asincroni, questo consente di sovrapporre il trasferimento di dati all'esecuzione dei kernel, migliorando l'efficienza in alcune situazioni.

Questa fase è essenziale, poiché ogni funzione kernel eseguita sulla GPU ha bisogno di accedere ai dati per operare. Tuttavia, una gestione non ottimale dei trasferimenti può comportare notevoli rallentamenti nel processo complessivo, rendendo cruciale il loro corretto utilizzo.

Esecuzione delle funzioni kernel e copia dei risultati all'host

Una volta che i dati sono stati trasferiti nella memoria del device, l'host lancia una funzione kernel sulla GPU. Il kernel, che rappresenta una funzione eseguita in parallelo da migliaia di thread, è il cuore del calcolo parallelo. La GPU organizza questi thread in blocchi e griglie, come specificato dall'host al momento del lancio, distribuendo il lavoro in modo efficiente e massimizzando le prestazioni dell'elaborazione parallela.

Terminato il calcolo sul device, i risultati vengono copiati dalla memoria della GPU alla memoria dell'host utilizzando `cudaMemcpy()` con il parametro opportuno:

```
cudaMemcpy(hostA, devA, size * sizeof(int), cudaMemcpyDeviceToHost);
```

Questo trasferimento consente all'host di analizzare i risultati o di proseguire con ulteriori elaborazioni. Poiché queste operazioni di copia possono rappresentare un collo di bottiglia, è cruciale gestirle in modo efficiente per evitare che incidano negativamente sulle prestazioni complessive. In alcuni casi, `cudaMemcpyAsync()` può essere utilizzata anche per questo trasferimento, sfruttando la sovrapposizione con altre operazioni per migliorare il throughput.

Deallocazione della memoria

Dopo il completamento dell'elaborazione, è essenziale deallocare la memoria allocata sia sull'host che sul device per evitare problemi di memoria o perdite di risorse. La deallocazione sul device si effettua utilizzando la funzione `cudaFree()`, che libera la memoria globale precedentemente riservata con `cudaMalloc()`. Analogamente, la memoria allocata sull'host deve essere liberata utilizzando la funzione standard `free()` in C, per recuperare le risorse del sistema. La deallocazione corretta di tutte le risorse, sia sull'host che sul device, è un passaggio fondamentale per garantire il funzionamento stabile delle applicazioni CUDA, soprattutto in programmi complessi o iterativi.

3.1.12 Architettura CUDA

L'architettura CUDA è progettata per massimizzare l'efficienza dell'elaborazione parallela, introducendo diverse tipologie di memorie e una struttura hardware ottimizzata per il calcolo massivo. Ogni tipologia di memoria ha uno scopo specifico e un utilizzo distinto, contribuendo al miglioramento delle prestazioni complessive [38] (*NVIDIA Corporation*, *2024*).

- **Memoria globale** (*Global memory*): Accessibile da tutti i thread del dispositivo, è la memoria principale della GPU. Sebbene offra una capacità elevata, ha una latenza significativa, rendendo necessario minimizzarne l'uso diretto e ottimizzarne gli accessi tramite strategie di *coalescing*, dove i thread accedono consecutivamente a posizioni di memoria contigue.
- **Memoria costante** (*Constant memory*): Ottimizzata per accessi simultanei e di sola lettura, è ideale per memorizzare parametri che non variano durante l'esecuzione del kernel. Quando tutti i thread accedono alla stessa variabile, la memoria costante garantisce prestazioni superiori rispetto alla memoria globale.
- **Memoria texture** (*Texture memory*): Progettata per accedere in modo efficiente a dati bidimensionali o con pattern regolari, è utile per applicazioni come il processamento di immagini. Grazie all'uso di unità hardware dedicate, può migliorare le prestazioni tramite interpolazioni e accessi ottimizzati.
- **Memoria condivisa** (*Shared memory*): Funziona come una cache locale per i thread dello stesso blocco, offrendo accessi molto più rapidi rispetto alla memoria globale. È ideale per la cooperazione tra thread di un blocco, riducendo la latenza delle operazioni e migliorando l'efficienza dei calcoli intra-blocco.

Utilizzo delle memorie in CUDA

La scelta della memoria da utilizzare dipende dal tipo di dati e dall'accesso richiesto dai thread, con ciascuna tipologia che offre vantaggi specifici:

- La **memoria globale** è flessibile e adatta per grandi quantità di dati, ma deve essere usata con cautela per evitare penalizzazioni dovute alla latenza.
- La **memoria costante** è particolarmente utile per dati statici e comuni a tutti i thread, garantendo un accesso rapido in questi scenari.

- La **memoria texture** eccelle nei casi in cui i dati richiedono interpolazioni o accessi spazialmente coerenti, sfruttando le sue ottimizzazioni hardware.
- La **memoria condivisa** riduce i costi di latenza per i dati condivisi tra thread dello stesso blocco, risultando particolarmente efficace in algoritmi cooperativi.

CUDA fornisce strumenti per dichiarare variabili nelle diverse memorie, come `__global__`, `__shared__`, e `__constant__`, definendo la visibilità e l'accessibilità dei dati. La comprensione e l'uso appropriato di queste memorie sono essenziali per ottimizzare le applicazioni.

L'architettura CUDA si basa su una struttura altamente parallela, dove i multiprocessori streaming (*Streaming Multiprocessors*, SMs) eseguono il lavoro dei thread. Ogni SM è composto da diversi *core CUDA*, che gestiscono le istruzioni assegnate. Questa configurazione permette di accelerare algoritmi su larga scala, massimizzando le prestazioni per operazioni ripetitive e parallele.

3.1.13 Librerie CUDA

CUDA offre un ricco ecosistema di librerie che permettono di sfruttare al meglio il potenziale delle GPU, semplificando il lavoro degli sviluppatori e accelerando applicazioni in svariati ambiti [39] (*NVIDIA Corporation*, *2024*). Queste librerie coprono una vasta gamma di esigenze: dall'algebra lineare e calcolo numerico, alle trasformate di Fourier e all'analisi di grafi. Ogni libreria è ottimizzata per sfruttare l'hardware della GPU, garantendo prestazioni elevate e una significativa riduzione dei tempi di sviluppo.

Tra le più utilizzate troviamo cuBLAS per operazioni di algebra lineare densa, cuFFT per le trasformate di Fourier, e strumenti più specializzati come cuSOLVER e cuSPARSE, pensati rispettivamente per lavorare con matrici dense e sparse. Oltre a queste librerie, in particolare, hanno avuto un ruolo centrale in questo progetto le librerie cuDSS e CUB, motivo per cui è importante introdurle prima di immergerci nella spiegazione di come abbiamo implementato in parallelo il Modello Cellulare di Potts.

cuSOLVER

Il toolkit cuSOLVER è una libreria di NVIDIA progettata per accelerare i calcoli di algebra lineare densa e sparsa su GPU. Fornisce un insieme di routine ottimizzate per operazioni come decomposizioni matriciali (LU, QR, Cholesky), fattorizzazioni, soluzioni di sistemi lineari e molto altro [36] (*NVIDIA Corporation*, *2024*). Queste funzioni sono ampiamente utilizzate in applicazioni scientifiche e ingegneristiche che richiedono calcoli ad alta precisione e performance elevate. Grazie alla stretta integrazione con altre librerie CUDA, come cuBLAS, cuSOLVER permette di sfruttare al massimo il parallelismo delle GPU per problemi di algebra lineare. Per utilizzare cuSOLVER, è necessario includere gli header appropriati: `<cusolverDn.h>` per le matrici dense, `<cusolverSp.h>` per le matrici sparse. Grazie a questa modularità, cuSOLVER offre flessibilità agli sviluppatori, consentendo di scegliere e combinare le funzionalità necessarie per ottimizzare le applicazioni in ambito scientifico e ingegneristico.

cuSPARSE

La libreria cuSPARSE, invece, è specificamente progettata per calcoli di algebra lineare su matrici sparse. Queste matrici, caratterizzate da un alto numero di elementi nulli, sono comunemente utilizzate in campi come simulazioni fisiche, analisi dei grafi e machine learning. cuSPARSE offre una serie di routine ottimizzate per operazioni come moltiplicazioni matrice-vettore (SpMV), risoluzione di sistemi lineari sparsi e conversioni tra formati di matrici sparse (ad esempio CSR e COO) [37] (*NVIDIA Corporation*, *2024*). Per accedere alle funzioni della libreria cuSPARSE, è necessario includere l'header `<cusparse.h>`. Questa libreria è progettata per ridurre al minimo l'uso di memoria e migliorare l'efficienza computazionale, sfruttando l'architettura parallela delle GPU. Grazie al supporto per diversi formati di rappresentazione, cuSPARSE rappresenta una soluzione versatile per la gestione efficiente di matrici sparse, adattandosi a numerosi contesti applicativi.

cuDSS

La libreria cuDSS (*CUDA Direct Sparse Solver*) è progettata specificamente per risolvere sistemi lineari sparsi con la possibilità di scegliere tra più algoritmi, tra cui le fattorizzazioni LU, QR e di Cholesky . Si tratta di un componente essenziale per applicazioni che richiedono alta efficienza e precisione nella gestione di matrici sparse, come simulazioni scientifiche, analisi strutturali e modelli computazionali. cuDSS si distingue per l'implementazione di algoritmi ottimizzati per l'architettura delle GPU, riducendo significativamente il tempo richiesto per la risoluzione di sistemi complessi. Inoltre, supporta diversi formati di rappresentazione delle matrici sparse, come CSR (*Compressed Sparse Row*), per garantire flessibilità e compatibilità con una vasta gamma di applicazioni. In questo progetto, cuDSS è stato fondamentale per accelerare la risoluzione dei sistemi lineari durante ogni passo di Monte Carlo, sfruttando la parallelizzazione della GPU per ridurre drasticamente i tempi di calcolo rispetto alla versione sequenziale.

CUB

CUB (*CUDA UnBound*) è una libreria altamente ottimizzata per il calcolo parallelo su GPU. Fornisce primitive di programmazione parallela fondamentali, come scansioni, riduzioni, ordinamenti e operazioni su array sparse. Queste primitive consentono agli sviluppatori di implementare operazioni personalizzate con un livello di efficienza elevato, sfruttando appieno l'architettura parallela delle GPU. CUB è progettata per essere modulare e facile da integrare, fornendo interfacce flessibili che si adattano a diversi scenari applicativi. Nel contesto di questo progetto, CUB è stata utilizzata per ottimizzare la gestione dei dati e alcune operazioni fondamentali, come il calcolo dei campi chimici ed elettrici, consentendo di migliorare ulteriormente le prestazioni complessive della simulazione. Grazie alla sua flessibilità, CUB rappresenta uno strumento essenziale per sviluppatori che desiderano ottenere il massimo dalla programmazione CUDA.

3.1.14 SLURM e la gestione dell'esecuzione parallela

SLURM (*Simple Linux Utility for Resource Management*) è un gestore di risorse open-source ampiamente utilizzato per ottimizzare l'allocazione e la gestione di risorse computazionali, come CPU, GPU e memoria. Questo strumento è particolarmente diffuso in ambienti di calcolo ad alte prestazioni (*High-Performance Computing, HPC*), grazie alla sua capacità di gestire in modo efficiente lavori paralleli e distribuiti su risorse condivise.

SLURM consente di pianificare ed eseguire lavori paralleli suddividendoli in *tasks* indipendenti, distribuiti in modo ottimale tra le risorse disponibili. Offre inoltre strumenti avanzati per il monitoraggio delle risorse e la personalizzazione dei parametri di esecuzione, garantendo un utilizzo bilanciato e adattabile alle necessità specifiche di ogni applicazione.

Grazie alla sua flessibilità, SLURM può essere utilizzato per una vasta gamma di applicazioni parallele, come simulazioni scientifiche, analisi di dati e addestramento di modelli di apprendimento automatico. Le sue funzionalità principali includono:

- **Allocazione dinamica delle risorse**: SLURM assegna CPU, GPU e memoria in base ai requisiti specificati dall'utente per ciascun lavoro.
- **Pianificazione avanzata**: consente di specificare dipendenze tra i lavori, priorità di esecuzione e limiti temporali.
- **Facilità di utilizzo**: permette di gestire l'esecuzione dei lavori tramite script `bash`, specificando le risorse richieste e i parametri di esecuzione.

L'utilizzo di SLURM consente di semplificare la gestione dell'esecuzione parallela, migliorando l'efficienza computazionale grazie a un'allocazione ottimale delle risorse. A seguire, viene presentato un esempio di script SLURM che potrebbe essere utilizzato per gestire un programma parallelo implementato in CUDA.

Esempio di script SLURM

Di seguito è riportato uno `script bash` esemplificativo per gestire l'esecuzione parallela di un programma CUDA utilizzando SLURM:

Listing 3.1: Esempio di script SLURM per l'esecuzione di un programma CUDA

```
#!/bin/bash
#SBATCH --job-name=cpm_cuda_job
#SBATCH --output=cpm_cuda_%j.out
#SBATCH --error=cpm_cuda_%j.err
#SBATCH --ntasks=1
#SBATCH --cpus-per-task=4
#SBATCH --mem=0
#SBATCH --gres=gpu:4
#SBATCH --partition=xgpu
#SBATCH --nodelist=gnode02

# Load CUDA module
module load cuda/11.6
```

```
# Run the CUDA program
srun ./cpm_cuda.out "$@"
```

Lo script inizia specificando i parametri principali tramite il comando `#SBATCH`, necessari per consentire a SLURM di gestire al meglio le risorse e lo scheduling del job. I parametri definiti includono:

- `--job-name`: il nome assegnato al job (`cpm_cuda_job`).
- `--output`: il file in cui viene salvato l'output standard del job (`cpm_cuda_%j.out`, dove `%j` rappresenta l'ID del job).
- `--error`: il file in cui viene salvato l'output degli errori (`cpm_cuda_%j.err`).
- `--ntasks`: il numero totale di task da eseguire (in questo caso `1`, corrispondente al job principale).
- `--cpus-per-task`: il numero di CPU allocate per ogni task (`4`).
- `--mem`: la quantità di memoria disponibile per il job (`0` indica che non ci sono limiti e viene utilizzata tutta la memoria disponibile).
- `--gres=gpu`: il numero di GPU richieste (`4`).
- `--partition`: la partizione specifica del cluster da utilizzare (`xgpu`).
- `--nodelist`: il nodo su cui eseguire il job (`gnode02`).

Successivamente, lo script carica il modulo contenente la versione di CUDA utilizzata nel programma (`11.6`) tramite il comando `module load`. Infine, il comando `srun` avvia il programma parallelo (`./cpm_cuda.out`) con eventuali parametri passati come argomenti grazie alla variabile `$@`. Questa specifica permette di personalizzare l'esecuzione su larga scala modificando parametri come la dimensione del reticolo, il numero iniziale di cellule e il numero di passi di Monte Carlo.

Questo esempio illustra come SLURM possa essere utilizzato per ottimizzare l'esecuzione di applicazioni parallele, gestendo automaticamente l'allocazione delle risorse e riducendo l'overhead operativo. SLURM rappresenta quindi uno strumento versatile ed efficace per migliorare l'efficienza delle simulazioni computazionali, indipendentemente dall'applicazione specifica.

Capitolo 4

Dall'implementazione parallela ai risultati

4.1 L'implementazione parallela

In questo capitolo viene presentata l'implementazione parallela del Modello Cellulare di Potts (CPM) utilizzando CUDA, un *framework* di calcolo ad alte prestazioni sviluppato da NVIDIA per sfruttare la potenza computazionale delle GPU. La transizione dal codice sequenziale al codice parallelo rappresenta un passo fondamentale per aumentare la scalabilità, consentendo di affrontare simulazioni su larga scala, che sarebbero proibitive utilizzando un'implementazione tradizionale su CPU.

Verranno analizzate le principali differenze tra il codice sequenziale, descritto nei capitoli precedenti, e la versione parallela, con particolare attenzione alla gestione delle strutture dati e al flusso di esecuzione. In seguito, saranno illustrate le parti chiave del codice che sono state parallelizzate: l'inizializzazione delle cellule del reticolo, la risoluzione dei sistemi lineari per il calcolo del campo chimico, l'aggiornamento delle cellule e la normalizzazione del campo chimico tramite algoritmi paralleli di *reduction*. L'analisi proseguirà con una discussione dei risultati ottenuti, confrontando le prestazioni in termini di velocità e accuratezza tra la versione sequenziale e quella parallela. Infine, verranno evidenziati i principali miglioramenti raggiunti, le difficoltà incontrate durante la transizione e le potenziali direzioni di sviluppo futuro per ottimizzare ulteriormente l'algoritmo [1].

4.1.1 Spiegazione dello pseudocodice parallelo

Il Modello Cellulare di Potts in parallelo descritto nello pseudocodice è suddiviso in tre fasi principali: inizializzazione, ciclo principale di simulazione e finalizzazione. Di seguito, ogni STEP è descritto nel dettaglio:

- **STEP 0: Inizializzazione**
  Questa fase prepara l'ambiente per la simulazione. Il reticolo (`lattice`) e i cam-

[1] L'implementazione parallela finale si distingue nettamente per prestazioni superiori rispetto alla versione sequenziale in C, mostrando miglioramenti evidenti già con matrici di piccole dimensioni e guadagni ancora più significativi su scala più ampia. Tuttavia, trattandosi di una tesi sperimentale, verranno anche discussi gli approcci che si sono rivelati inefficaci durante il percorso di sviluppo. Questi tentativi, pur non essendo direttamente produttivi, sono stati passi fondamentali per spingere la ricerca fino al raggiungimento di quello che si è rivelato essere l'efficiente risultato finale.

pi chimici (`chemical`) vengono inizializzati con valori predefiniti. I parametri CUDA, come il numero di blocchi e thread, sono configurati, e viene allocata la memoria necessaria sia sulla CPU (host) che sulla GPU (device).

- **STEP 1: Imposta l'ambiente**
  Lo stato iniziale del reticolo L^0 è configurato per rappresentare la distribuzione iniziale delle celle. I campi elettrici, se utilizzati, vengono allocati e inizializzati. In questa fase, vengono configurati anche i gestori CUDA necessari per la risoluzione di sistemi lineari, sia sparsi che densi, utilizzando librerie come `cuSolver` e `cuSparse`.

- **STEP 2: Ciclo principale della simulazione (passi di Monte Carlo)**
  Questa è la fase centrale del modello, iterata per NUM_MCS passi Monte Carlo:

  – Ogni 10 passi, lo stato corrente del reticolo e dei campi chimici viene salvato in `lattice_history` e `chemical_history`, rispettivamente, per analisi e visualizzazione.

  – Per ogni pixel del reticolo:

  * Un pixel casuale P viene selezionato come "sorgente", e un suo vicino casuale Q come "target".
  * Si calcola ΔH_{total}, che rappresenta il cambiamento totale di energia del sistema considerando vari contributi, come adesione, volume, superficie, chemotassi e campi elettrici.
  * L'algoritmo di Metropolis viene applicato per decidere se accettare la transizione da P a Q, sulla base di ΔH_{total} e della temperatura T.
  * Se la transizione è accettata, il reticolo e i volumi delle celle vengono aggiornati.

  – Dopo l'aggiornamento del reticolo, viene risolto il sistema lineare per calcolare i campi elettrici, sfruttando le librerie CUDA.

  – Infine, i campi chimici vengono aggiornati e normalizzati tramite un'operazione di reduction parallela.

- **STEP 3: Finalizzazione**
  In questa fase, viene deallocata la memoria utilizzata sia sull'host che sul device, assicurando che le risorse siano rilasciate correttamente e che il sistema sia pronto per successive esecuzioni.

Algoritmo 4: Modello Cellulare di Potts (CUDA)

Input:

`LATTICE_SIZE`	Dimensione del reticolo
`NUM_CELLS`	Numero di celle
`NUM_MCS`	Numero di passi Monte Carlo
`T`	Temperatura

Output:

`lattice_history`	Storico degli stati del reticolo
`chemical_history`	Storico degli stati del campo chimico

Function *main*:
- **STEP 0: Inizializzazione** ;
- Inizializza il reticolo e i campi chimici;
- Imposta i parametri CUDA e alloca la memoria;
- **STEP 1: Imposta l'ambiente** ;
- Imposta lo stato iniziale del reticolo L^0;
- Alloca e inizializza i campi elettrici;
- Configura i gestori CUDA per i sistemi sparsi e densi;
- **STEP 2: Ciclo principale della simulazione (passi di Monte Carlo)**;
- **for** $mcs = 1, \ldots, NUM\_MCS$ **do**
  - **if** $mcs \bmod 10 = 0$ **then**
    - Salva lo stato corrente del reticolo e dei campi chimici in history;
  - **foreach** *pixel nel reticolo* **do**
    - Seleziona un pixel casuale P sul reticolo come "sorgente";
    - Seleziona un vicino casuale Q di P come "target";
    - Calcola ΔH_{total} includendo i contributi di $H_{adhesion}$, H_{volume}, $H_{surface}$, $H_{chemotaxis}$ e $H_{electric}$;
    - Applica l'algoritmo di Metropolis con probabilità basata su ΔH_{total} e T;
    - **if** *la transizione è accettata* **then**
      - Aggiorna il reticolo e i volumi delle celle;
  - Risolvi il sistema per il campo elettrico;
  - Aggiorna i campi chimici e normalizza;
- **STEP 3: Finalizzazione**;
- Dealloca la memoria utilizzata;

Motivazioni del passaggio al parallelo

L'implementazione parallela del Modello Cellulare di Potts rappresenta un'evoluzione significativa, motivata dalla necessità di affrontare simulazioni su larga scala e di sfruttare appieno la potenza computazionale delle GPU. Grazie alla nuova versione parallela basata su CUDA, è stato possibile raggiungere risultati eccezionali in termini di velocità, efficienza e scalabilità, superando nettamente le prestazioni del codice sequenziale in C. Le motivazioni principali alla base di questo passaggio includono:

1. **Gestione delle matrici**: Il codice sequenziale, pur eccellente nella gestione di matrici sparse, mostra limiti significativi quando si affrontano simulazioni dense. L'uso di LAPACK su CPU, infatti, comporta elevati requisiti di memoria e tempi

di esecuzione proibitivi per reticoli di grandi dimensioni. La nuova implementazione CUDA, invece, gestisce le matrici dense con una rapidità e un'efficienza impareggiabili, eliminando i colli di bottiglia del calcolo su CPU.

2. **Ottimizzazione delle simulazioni sparse**: Sebbene il codice sequenziale in C basato su SuiteSparse offra ottime prestazioni per sistemi sparsi, la versione parallela in CUDA ha introdotto una fattorizzazione LU efficiente, garantendo velocità di esecuzione straordinarie, nettamente superiori sia alla controparte sequenziale che a MATLAB. Questo risultato rappresenta un'importante pietra miliare per il CPM, consentendo di analizzare configurazioni più complesse e più ampie rispetto al passato.

3. **Scalabilità e parallelismo**: L'implementazione parallela sfrutta le potenzialità della GPU distribuendo il carico computazionale su migliaia di thread, permettendo simulazioni su reticoli di grandi dimensioni con tempi drasticamente ridotti. Questa scalabilità non solo accelera la risoluzione di sistemi lineari, ma rende anche possibile affrontare problemi che altrimenti sarebbero stati irrisolvibili su CPU.

4. **Opportunità per innovazione e sviluppo futuro**: L'adozione di CUDA apre nuove prospettive per integrare metodi avanzati di ottimizzazione e analisi, come algoritmi specializzati per la risoluzione di sistemi lineari o strategie per il calcolo di energie aggiuntive. Inoltre, la solida base parallela costruita consente di affrontare simulazioni ancora più ambiziose in futuro, spingendo i limiti della ricerca computazionale nel campo dell'angiogenesi e del CPM.

Confronto tra strutture dati

Le principali differenze nella gestione delle strutture dati tra il codice sequenziale e quello parallelo riguardano la rappresentazione delle matrici e l'allocazione della memoria.

- **Da matrici 2D a array 1D**:
  Nel codice sequenziale, le matrici del reticolo, campo chimico e campo elettrico (`lattice, chemical, E_field_x, E_field_y`) sono rappresentate come puntatori doppi (`int**, double**`), permettendo un accesso diretto agli elementi tramite indici `[i][j]`. Nel codice parallelo, queste strutture sono state convertite in array 1D (`int*, double*`) per facilitarne la gestione su GPU, dove l'accesso agli elementi avviene tramite un unico indice `i*L+j`.

- **Gestione della memoria tra host e device**:
  Nel codice sequenziale, tutta la memoria è allocata sulla CPU tramite funzioni come malloc. Nel codice parallelo, i dati sono divisi tra memoria host (CPU) e memoria device (GPU). La memoria su GPU viene allocata con cudaMalloc e i dati vengono trasferiti tra host e device utilizzando cudaMemcpy. Per esempio: Il reticolo `lattice` è rappresentato da `h_lattice` su host e `d_lattice` su device. Il campo chimico (`chemical`) è suddiviso in `h_chemical` e `d_chemical`. Buffer aggiuntivi (`d_new_chemical`) sono utilizzati per operazioni intermedie su GPU.

- **Utilizzo di descrittori e handler per cuDSS**:
  L'uso di cuDSS per operazioni avanzate, come la risoluzione di sistemi lineari o la fattorizzazione LU, richiede l'inizializzazione di handler e descrittori specifici.
  Gli handler (`cudssHandle_t`) gestiscono il contesto e la configurazione delle operazioni eseguite tramite cuDSS.
  I descrittori (`cudssStatus_t`, `cudssMatrix_t`) rappresentano le proprietà e i metadati delle matrici e dei vettori utilizzati nei calcoli. Ad esempio, una matrice può essere descritta specificando il tipo di struttura (densa, sparsa), il formato (COO, CSR), e le dimensioni, mentre i vettori sono descritti con la loro lunghezza e il tipo di dati. Questi descrittori vengono passati alle funzioni di cuDSS per eseguire operazioni ottimizzate direttamente sulla GPU. Questo livello di astrazione è essenziale per sfruttare appieno le capacità della libreria, riducendo l'interazione manuale con i dati grezzi e migliorando la gestione delle risorse computazionali.

Questa transizione da sequenziale a parallelo ha richiesto un adattamento del codice per garantire un accesso efficiente ai dati, evitando copie inutili tra CPU e GPU.

Confronto tra flussi di esecuzione

Il flusso di esecuzione nel codice parallelo differisce significativamente da quello sequenziale a causa della natura del calcolo distribuito e della gestione della memoria tra CPU e GPU. In questa sezione vengono analizzate le principali differenze.

Flusso di esecuzione nel codice sequenziale

Nel codice sequenziale, tutti i calcoli vengono eseguiti sulla CPU, seguendo un approccio iterativo. Il ciclo principale, che implementa l'algoritmo Monte Carlo, elabora ogni cella del reticolo una alla volta. Le operazioni chiave, come il calcolo dell'energia di transizione e l'aggiornamento del campo chimico, sono implementate con cicli annidati che iterano sull'intero reticolo. L'approccio sequenziale soffre di limiti di scalabilità, in quanto la complessità cresce rapidamente con l'aumentare delle dimensioni del reticolo e del numero di passi Monte Carlo (*NUM_MCS*).

Flusso di esecuzione nel codice parallelo

Nel codice parallelo, le operazioni principali vengono suddivise tra migliaia di thread GPU utilizzando kernel CUDA. Ogni kernel è progettato per eseguire calcoli specifici su porzioni del reticolo in modo simultaneo. Le principali caratteristiche del flusso parallelo includono:

- **Distribuzione del carico**: Ogni thread GPU è responsabile di elaborare un singolo elemento del reticolo o una porzione di esso. Ad esempio, nella normalizzazione del campo chimico, i thread vengono distribuiti su righe o colonne del reticolo.

- **Sincronizzazione**: Poiché i calcoli coinvolgono dati condivisi tra più thread, è necessaria una sincronizzazione esplicita per garantire la coerenza dei risultati. Funzioni come `__syncthreads()` vengono utilizzate nei kernel CUDA per sincronizzare i thread all'interno dello stesso blocco.

- **Memoria device e trasferimenti dati**: I dati vengono trasferiti dalla memoria host (CPU) alla memoria device (GPU) prima dell'elaborazione, e i risultati vengono copiati nuovamente su host al termine delle operazioni. Questo introduce un overhead aggiuntivo rispetto all'esecuzione sequenziale.

Confronto diretto

Le principali differenze tra i due approcci possono essere riassunte come segue:

1. **Granularità del calcolo**:
   Nel codice sequenziale, ogni operazione viene eseguita iterativamente su ogni elemento del reticolo, attraverso cicli annidati che gestiscono l'intero insieme di celle. Al contrario, nel codice parallelo, le operazioni principali, come l'inizializzazione delle cellule, l'aggiornamento del campo chimico e la risoluzione dei sistemi lineari, vengono distribuite tra i thread della GPU, sfruttando appieno la potenza computazionale delle unità di calcolo parallelo. Nonostante ciò, alcune parti dell'algoritmo, come il calcolo dell'energia totale o la sincronizzazione degli stati tra celle vicine, mantengono una componente iterativa a causa della natura stocastica dell'algoritmo di Metropolis e del Metodo di Monte Carlo, che richiedono una dipendenza sequenziale tra i passi del ciclo.

2. **Efficienza del calcolo**:
   La parallelizzazione consente di ottenere un enorme guadagno di efficienza, in particolare su problemi di larga scala. Operazioni come l'inizializzazione delle cellule e l'aggiornamento del campo chimico diventano progressivamente più vantaggiose all'aumentare delle dimensioni del reticolo, evidenziando il vantaggio della GPU rispetto al calcolo sequenziale su CPU. Per problemi di dimensioni molto ridotte, l'overhead associato alla gestione della memoria e alla sincronizzazione tra host e device può temporaneamente ridurre i benefici della parallelizzazione, ma già con un lieve incremento delle dimensioni del problema, le operazioni parallele iniziano a mostrare una significativa superiorità rispetto al codice sequenziale.

3. **Scalabilità e note d'interesse**:
   Le GPU eccellono su larga scala, dove la gestione parallela di operazioni computazionalmente intense diventa incredibilmente efficiente rispetto al calcolo sequenziale. Questo aspetto è particolarmente evidente per operazioni come l'aggiornamento del campo chimico e la gestione di reticoli di grandi dimensioni. Tali risultati dimostrano il potenziale straordinario delle GPU nell'affrontare simulazioni su larga scala, rendendo il codice parallelo non solo più performante, ma anche scalabile e adatto a scenari più complessi.

4. **Limitazioni apparenti**:
   Più che vere limitazioni, è interessante notare che per problemi di dimensioni estremamente ridotte, l'overhead iniziale può rendere il calcolo sequenziale momentaneamente più efficiente. Tuttavia, con l'aumento anche minimo delle dimensioni del problema, il codice parallelo supera rapidamente queste inefficienze, mostrando la capacità della GPU di trasformare l'aumento della complessità in un'opportunità per massimizzare le prestazioni.

Considerazioni finali

L'implementazione parallela offre vantaggi significativi per simulazioni su una grande quantità di dati e rappresenta un passo importante verso un'ulteriore futura ottimizzazione dell'algoritmo.

4.1.2 Parallelizzazione delle parti chiave del codice

La parallelizzazione del codice sequenziale ha richiesto una riorganizzazione delle parti principali dell'algoritmo per sfruttare appieno la potenza computazionale della GPU. In questa sezione, vengono descritte le quattro componenti fondamentali che sono state parallelizzate:

1. **Inizializzazione delle cellule del reticolo**: una fase cruciale per la configurazione iniziale del sistema, che determina lo stato di partenza delle simulazioni.

2. **Risoluzione del sistema lineare**: un'operazione essenziale per il calcolo del campo chimico, che richiede una gestione efficiente delle matrici sparse o dense.

3. **Aggiornamento del campo chimico**: una fase iterativa in cui i valori del campo chimico vengono aggiornati per riflettere le interazioni locali e l'equilibrio dinamico del sistema.

4. **Normalizzazione del campo chimico**: una fase in cui il campo chimico viene scalato utilizzando una reduction parallela per trovare il massimo valore.

Per ciascuna di queste parti, verranno forniti pseudocodici che illustrano l'algoritmo utilizzato, seguiti da una spiegazione dettagliata delle operazioni svolte.

4.1.3 Inizializzazione delle cellule del reticolo

L'inizializzazione del reticolo permette di configurare lo stato iniziale della simulazione. Ogni cellula deve essere distribuita uniformemente sul reticolo, rispettando il volume target assegnato. Questo garantisce che le condizioni iniziali siano corrette e che il sistema sia pronto per l'esecuzione CPM.

Nel contesto di una simulazione parallela, è necessario distribuire l'inizializzazione tra CPU e GPU per ottimizzare le prestazioni. La procedura implementata si compone di tre fasi principali:

- **Generazione di una lista di siti pre-allocati sulla CPU**: Questa lista specifica le posizioni iniziali delle cellule, distribuite casualmente sul reticolo.

- **Trasferimento della lista sulla GPU e assegnazione dei siti**: Tramite un kernel CUDA, ogni cellula viene assegnata ai siti specificati nella lista. Per garantire consistenza e non sovrascrivere dati, vengono utilizzate operazioni atomiche.

- **Copia dei risultati dal device (GPU) all'host (CPU)**: Una volta completata l'inizializzazione, i dati aggiornati del reticolo e dei volumi delle cellule vengono trasferiti sull'host per ulteriori elaborazioni.

Questa procedura è stata progettata per bilanciare il carico computazionale e minimizzare i conflitti di memoria durante l'assegnazione delle cellule.

Funzione generate_site_list

La funzione `generate_site_list` è responsabile della generazione della lista di siti in cui le cellule saranno posizionate inizialmente. Questa operazione viene svolta sulla CPU. Di seguito, la spiegazione riga per riga:

- Riga 1: La funzione prende come input la lista `site_list`, la dimensione del reticolo (`size`), il numero di cellule (`num_cells`) e il volume target per cellula (`target_volume`). Questi parametri determinano la distribuzione iniziale delle cellule sul reticolo.
- Riga 3: Viene calcolato il numero totale di siti necessari per contenere tutte le cellule, dato da `num_cells` $\times$ `target_volume`.
- Righe 5-7 Un ciclo `for` viene eseguito per generare casualmente gli indici dei siti all'interno del reticolo. La funzione `random` restituisce un valore compreso tra 0 e `size`^{2}-1, che rappresenta un indice valido per un sito.
- Alla fine dell'algoritmo, la lista `site_list` è popolata con gli indici generati.

Algoritmo 5: Genera lista di siti (inizializzazione delle cellule)

Input:

`site_list`	Array per memorizzare i siti allocati
`size`	Dimensione del reticolo
`num_cells`	Numero di cellule
`target_volume`	Volume target per cellula

Output:

`site_list`	Array popolato con siti pre-allocati

```
1 function generate_site_list(site_list, size, num_cells, target_volume):
2     STEP 1: Calcola il numero totale di siti
3     total_sites ← num_cells × target_volume
4     STEP 2: Popola la lista dei siti con indici casuali
5     for i ← 0 to total_sites − 1 do
6         site_list[i] ← random(0, size × size − 1)
7         // Assegna un indice casuale
```

Funzione assign_cells_kernel

La funzione `assign_cells_kernel` viene eseguita in parallelo sulla GPU e si occupa di assegnare le cellule ai siti pre-allocati nel reticolo. Utilizza un kernel CUDA per sfruttare il parallelismo intrinseco della GPU e gestisce l'assegnazione in modo sicuro grazie alle operazioni atomiche. Di seguito, la spiegazione dettagliata riga per riga:

- Righe 2-3: L'indice globale del thread `idx` viene calcolato combinando l'indice del blocco (`blockIdx.x`) e l'indice del thread (`threadIdx.x`). Questo indice identifica quale thread è responsabile di un determinato sito nella lista pre-allocata.
- Righe 4-6: Se `idx` è maggiore o uguale al numero totale di siti (`num_cells` $\times$ `target_volume`), il thread termina immediatamente, poiché non ci sono più siti da elaborare.

- Righe 7-8: L'ID della cellula corrente (`cell`) viene calcolato dividendo l'indice del thread (`idx`) per il volume target (`target_volume`). Viene poi aggiunto 1 per ottenere un ID basato su 1 (invece che 0).
- Righe 9: Viene recuperato l'indice del sito pre-allocato dalla lista `site_list`, corrispondente all'indice del thread.
- Righe 10-11: Le coordinate del sito nel reticolo (`x`, `y`) vengono calcolate dividendo l'indice del sito (`site_idx`) per la dimensione del reticolo (`size`) e calcolando il resto della divisione. Questi calcoli servono a convertire l'indice lineare in coordinate bidimensionali.
- Righe 12-14: L'operazione `atomicCAS` (Compare-And-Swap) tenta di assegnare la cellula al sito solo se il valore attuale del sito è 0 (vuoto). Questa operazione garantisce che due thread non sovrascrivano lo stesso sito contemporaneamente. Se l'assegnazione ha successo, il volume della cellula viene incrementato tramite un'operazione atomica `atomicAdd`.

Algoritmo 6: Assegna le cellule ai siti pre-allocati (inizializzazione delle cellule)

Input:

`lattice`	Array 1D del reticolo sul device
`site_list`	Lista di siti pre-allocati sul device
`cell_volumes`	Array dei volumi delle cellule sul device
`size`	Dimensione del reticolo
`target_volume`	Volume target per cellula
`num_cells`	Numero di cellule

Output:

Aggiornamento degli array lattice e cell_volumes sul device

1 **kernel** *assign_cells_kernel(lattice, site_list, cell_volumes, size, target_volume, num_cells)*:
2 **STEP 1:** Calcola l'indice globale del thread
3 $idx \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
4 **STEP 2:** Verifica se l'indice del thread è nei limiti
5 **if** $idx \geq num\_cells \times target\_volume$ **then**
6 **return**
7 **STEP 3:** Calcola l'ID della cellula e le coordinate del sito
8 $cell \leftarrow floor(idx/target\_volume) + 1$ Determina la cellula corrente
9 $site\_idx \leftarrow site\_list[idx]$ Ottieni l'indice del sito
10 $x \leftarrow floor(site\_idx/size)$ Calcola la coordinata x
11 $y \leftarrow site\_idx \bmod size$ Calcola la coordinata y
12 **STEP 4:** Assegna la cellula al sito se vuoto
13 **if** $atomicCAS(lattice[x \times size + y], 0, cell) == 0$ **then**
14 $atomicAdd(cell\_volumes[cell], 1)$ Incrementa il volume della cellula

Funzione initialize_cells_cuda

La funzione `initialize_cells_cuda` coordina l'intero processo di inizializzazione del reticolo utilizzando CUDA. La procedura combina la generazione della lista di

siti su CPU, il trasferimento dei dati alla GPU, e l'esecuzione del kernel CUDA per assegnare le cellule ai siti. Al termine, i risultati vengono copiati dalla GPU alla CPU per ulteriori elaborazioni. Di seguito, la spiegazione dettagliata riga per riga:

- Righe 2-3: Viene calcolato il numero totale di siti necessari per contenere tutte le cellule, dato da `num_cells` $\times$ `target_volume`.
- Riga 5: Viene allocata memoria sull'host per la lista dei siti (`h_site_list`) utilizzando la funzione `malloc`. Questa lista conterrà gli indici dei siti generati casualmente.
- Riga 6: Viene allocata memoria sul device per la lista dei siti (`d_site_list`) utilizzando la funzione `cudaMalloc`. Questo passaggio garantisce che i dati siano pronti per l'elaborazione sulla GPU.
- Riga 8: La funzione `generate_site_list` viene chiamata per popolare la lista dei siti sull'host. Gli indici generati casualmente indicano le posizioni iniziali delle cellule.
- Riga 9: La lista dei siti viene trasferita dalla memoria host alla memoria device utilizzando `cudaMemcpy`. Questo permette alla GPU di utilizzare i dati durante l'esecuzione del kernel.
- Righe 10-12: Si calcola il numero di blocchi e thread necessari per eseguire il kernel CUDA. I thread per blocco sono fissati a 256, e il numero di blocchi viene calcolato come il rapporto tra il numero totale di siti e i thread per blocco, arrotondato per eccesso.
- Riga 13: Il kernel `assign_cells_kernel` viene lanciato con la configurazione di blocchi e thread calcolata. Questo kernel assegna le cellule ai siti pre-allocati nel reticolo.
- Riga 14: La funzione `cudaDeviceSynchronize` viene chiamata per assicurarsi che tutti i thread GPU abbiano completato il loro lavoro prima di procedere.
- Righe 15-17: I dati aggiornati (reticolo e volumi delle cellule) vengono copiati dalla memoria device alla memoria host utilizzando `cudaMemcpy`, rendendoli disponibili per ulteriori elaborazioni sulla CPU.
- Righe 18-20: La memoria allocata per la lista dei siti viene liberata sia sull'host (`free`) che sul device (`cudaFree`).

Algoritmo 7: Inizializzazione delle cellule

Input:

`d_lattice`	Array del reticolo sul device
`h_lattice`	Array del reticolo sull'host
`d_cell_volumes`	Array dei volumi delle cellule sul device
`h_cell_volumes`	Array dei volumi delle cellule sull'host
`target_volume`	Volume target per cellula
`num_cells`	Numero di cellule
`size`	Dimensione del reticolo

Output:

Array lattice e cell_volumes inizializzati

function *initialize_cells_cuda(d_lattice, h_lattice, d_cell_volumes, h_cell_volumes, target_volume, num_cells, size)*:

- **STEP 1:** Calcola il numero totale di siti
- $total\_sites \leftarrow num\_cells \times target\_volume$
- **STEP 2:** Alloca la memoria
- $h\_site\_list \leftarrow malloc(total\_sites)$
- $d\_site\_list \leftarrow cudaMalloc(total\_sites)$
- **STEP 3:** Popola e copia la lista dei siti
- $generate\_site\_list(h\_site\_list, size, num\_cells, target\_volume)$
- $cudaMemcpy(d\_site\_list, h\_site\_list, total\_sites)$
- **STEP 4:** Lancia il kernel per l'assegnazione delle celle
- $threads\_per\_block \leftarrow 256$
- $blocks\_per\_grid \leftarrow ceil(total\_sites/threads\_per\_block)$
- assign_cells_kernel$<<<$ $blocks\_per\_grid, threads\_per\_block >>>$ $(d\_lattice, d\_site\_list, d\_cell\_volumes, size, target\_volume, num\_cells)$
- $cudaDeviceSynchronize()$
- **STEP 5:** Copia i risultati sull'host
- cudaMemcpy($h\_lattice$, $d\_lattice$, $size \times size$)
- cudaMemcpy($h\_cell\_volumes$, $d\_cell\_volumes$, $num\_cells + 1$)
- **STEP 6:** Libera la memoria allocata
- free(h_site_list)
- cudaFree(d_site_list)

4.1.4 Risoluzione del sistema lineare

La risoluzione del sistema lineare rappresenta la parte più onerosa e critica dell'intero algoritmo, richiedendo un'implementazione ottimizzata per affrontare le dimensioni del problema in modo efficiente. Considerando un reticolo standard di dimensione 200×200, la matrice A del sistema lineare ha dimensioni pari a $N \cdot N \times N \cdot N$, dove $N = 200$. Ciò significa che A è una matrice di dimensione 40000×40000, contenente 1.6×10^9 elementi, mentre il vettore b ha dimensione 40000.

Date queste dimensioni, l'utilizzo di una rappresentazione densa per la matrice A risulta estremamente oneroso in termini di memoria e tempo computazionale. Per questo motivo, si adotta una rappresentazione sparsa, che consente di lavorare esclusivamente con i valori non nulli, riducendo drasticamente le risorse richieste.

Tuttavia, per confronto, è stata implementata una risoluzione lineare anche con matrice densa, sia in modalità sequenziale che parallela.

I risultati mostrano chiaramente la superiorità del codice parallelo: l'esecuzione parallela con matrice densa supera nettamente quella sequenziale, dimostrando i vantaggi dell'utilizzo delle GPU per problemi computazionalmente intensi. Allo stesso tempo, la risoluzione parallela con matrice sparsa non solo supera la versione MATLAB originale, ma anche il codice sequenziale in C, confermando l'efficienza dell'approccio ottimizzato per GPU.

Il miglioramento delle prestazioni del codice parallelo è attribuibile a una combinazione di fattori: la distribuzione efficace delle operazioni tra i thread GPU, la gestione ottimizzata della memoria tramite rappresentazioni sparse, e la capacità della GPU di scalare con l'aumento delle dimensioni del problema. Anche i costi di overhead associati al trasferimento dei dati tra CPU e GPU, che in passato rappresentavano un limite per dei tentativi precedenti, sono ampiamente compensati dai guadagni computazionali offerti dalla parallelizzazione.

Infine, è importante sottolineare che la natura scalabile delle GPU si dimostra cruciale in scenari con dimensioni crescenti del reticolo. Con incrementi nella dimensione del reticolo, la matrice A cresce esponenzialmente, ma grazie all'approccio parallelo, il sistema riesce a gestire questa complessità in modo efficiente, rendendo possibile l'esecuzione di simulazioni su larga scala con risultati altamente performanti.

Il ruolo di cuSOLVER e cuSPARSE

Le librerie `cuSOLVER` e `cuSPARSE` hanno un ruolo fondamentale nella risoluzione del sistema lineare, offrendo funzioni ottimizzate per la gestione di matrici dense e sparse sulla GPU. Si comincia con la creazione degli *handler*, strutture che permettono di interfacciarsi con le funzioni delle librerie CUDA.

La creazione degli *handler* avviene mediante chiamate specifiche:

- `cusolverDnCreate()` per inizializzare un *handler* per la risoluzione di sistemi lineari densi.
- `cusolverSpCreate()` per inizializzare un *handler* per sistemi lineari sparsi.

Al termine delle operazioni, è necessario distruggere gli *handler* per liberare risorse, utilizzando rispettivamente `cusolverDnDestroy()` e `cusolverSpDestroy()`.

Per la fattorizzazione LU e la risoluzione di sistemi lineari densi, le funzioni principali utilizzate sono:

- `cusolverDnDgetrf_bufferSize()` per calcolare la dimensione del buffer necessaria alla fattorizzazione LU.
- `cusolverDnDgetrf()` per eseguire la fattorizzazione LU.
- `cusolverDnDgetrs()` per risolvere il sistema lineare dato il risultato della fattorizzazione LU.

Per un primo tentativo verso la parallelizzazione del codice per la gestione delle matrici sparse, sono state usate le librerie di cuSOLVER e cuSPARSE e le funzioni principali utilizzate sono:

- `cusparseCreateMatDescr()` per creare una descrizione della matrice.

- `cusparseSetMatType()` per definire il tipo di matrice (ad esempio simmetrica o generale).
- `cusparseSetMatIndexBase()` per impostare l'indice base della matrice (0 o 1).
- `cusparseDestroyMatDescr()` per distruggere la descrizione della matrice.
- `cusolverSpDcsrlsvqr()` per risolvere il sistema lineare sparso utilizzando la fattorizzazione QR.

Queste funzioni permettono di sfruttare al meglio le capacità di parallelizzazione della GPU per la risoluzione di sistemi lineari, adattandosi sia a matrici dense che sparse.

È bene notare però, che nella versione definitiva del progetto, cuSOLVER e cuSPARSE sono usati solo per le matrici dense. L'introduzione di cuDSS ha dimostrato prestazioni superiori, surclassando le precedenti implementazioni per matrici sparse e affermandosi come la soluzione più adatta per i requisiti del progetto.

Fattorizzazione QR

In una delle prime versioni del progetto parallelo, è stata applicata una risoluzione del sistema lineare tramite fattorizzazione QR.
La fattorizzazione QR è una tecnica che decompone una matrice A in un prodotto $A = QR$, dove Q è una matrice ortogonale (o unitari nel caso complesso) e R è una matrice triangolare superiore. Questa fattorizzazione è particolarmente utile per la risoluzione di sistemi lineari sovradeterminati e per problemi di regressione [19] (*Gene H Golub e Charles F Van Loan*, *2013*).

Il calcolo della fattorizzazione QR può essere eseguito attraverso due metodi principali:

- **Trasformazioni di Householder**: queste trasformazioni generano matrici ortogonali che eliminano iterativamente gli elementi al di sotto della diagonale [50] (*Lloyd N Trefethen e David Bau III*, *1997*).
- **Rotazioni di Givens**: utilizzate per annullare specifici elementi della matrice tramite rotazioni nel piano, con un approccio più localizzato rispetto a Householder [14] (*James W Demmel*, *1997*).

In ambito parallelo, le trasformazioni di Householder sono spesso preferite per la loro efficienza computazionale, poiché si prestano meglio alla parallelizzazione su GPU [47] (*NVIDIA Documentation Team*, *2020*). La fattorizzazione QR offre una maggiore stabilità numerica rispetto alla fattorizzazione LU, rendendola particolarmente adatta per matrici mal condizionate o sparse. Inoltre, la QR preserva le proprietà ortogonali, semplificando l'analisi numerica dei sistemi lineari risolti con questa tecnica [24] (*Nicholas J Higham*, *2002*).

cuDSS: una nuova frontiera nella risoluzione di sistemi lineari sparsi

NVIDIA ha recentemente introdotto cuDSS (*CUDA Direct Sparse Solver*), una libreria ottimizzata per la risoluzione di sistemi lineari con matrici sparse direttamente

sulla GPU [10] (*NVIDIA Corporation*, *2023*). Questa libreria è progettata per sfruttare al meglio le capacità di parallelismo delle GPU NVIDIA, offrendo prestazioni significativamente superiori rispetto ai solver basati esclusivamente su CPU.

CuDSS supporta l'esecuzione su piattaforme single-GPU, multi-GPU e multi-nodo, rendendola altamente scalabile per applicazioni di grandi dimensioni. Inoltre, è ottimizzata per le architetture GPU più recenti, come il Grace Hopper Superchip [12] (*NVIDIA Corporation*, *2023*), garantendo compatibilità e prestazioni elevate su una vasta gamma di dispositivi NVIDIA.

Essendo una libreria di prima generazione, cuDSS è ancora in fase di sviluppo e miglioramento. Attualmente, è disponibile in versione preview, il che implica che l'API potrebbe subire modifiche nelle future release [11] (*NVIDIA Corporation*, *2023*).

L'integrazione di cuDSS nel progetto ha permesso di superare le limitazioni delle precedenti implementazioni basate su cuSOLVER e cuSPARSE, offrendo una soluzione più efficiente per la risoluzione di sistemi lineari con matrici sparse. Questo ha portato a un notevole incremento delle prestazioni, rendendo cuDSS la scelta preferita per le esigenze computazionali del progetto.

All'interno del progetto, l'integrazione di cuDSS ha richiesto l'utilizzo di specifiche funzioni e strutture di supporto:

- **Gestione degli handler**:

- `cudssHandle_t`: rappresenta l'handle principale per l'interazione con la libreria cuDSS.

- `cudssConfig_t`: contiene le impostazioni di configurazione del solver, permettendo di definire parametri specifici per la risoluzione del sistema.

- `cudssData_t`: gestisce i dati necessari durante le operazioni del solver, facilitando l'accesso e la manipolazione delle informazioni durante il processo di risoluzione.

- **Creazione e gestione delle matrici**:

- `cudssMatrix_t`: rappresenta le matrici sparse e dense utilizzate nel sistema, con funzioni dedicate per la loro creazione e gestione. `Fasi di esecuzione`: la risoluzione del sistema lineare con cuDSS è suddivisa in diverse fasi, ciascuna gestita da apposite funzioni.

- `cudssExecute()`: esegue operazioni specifiche come l'analisi simbolica, la fattorizzazione numerica e la risoluzione del sistema, a seconda del parametro di fase fornito.

Un aspetto distintivo di cuDSS è la possibilità di selezionare il tipo di fattorizzazione da utilizzare nella risoluzione del sistema lineare. Attraverso l'impostazione di specifici flag nel `cudssConfig_t`, è possibile scegliere tra diverse metodologie di fattorizzazione, come LU, Cholesky, o QR, adattando il processo alle caratteristiche specifiche del problema da risolvere.

L'integrazione di cuDSS nel progetto ha permesso di superare le limitazioni delle precedenti implementazioni basate su `cuSOLVER` e `cuSPARSE`, offrendo una soluzione più efficiente per la risoluzione di sistemi lineari con matrici sparse. Questo ha portato a un notevole incremento delle prestazioni, rendendo cuDSS la scelta preferita per le esigenze computazionali del progetto.

Funzione cuda_build_dense_linear_system

La funzione `cuda_build_dense_linear_system` ha il compito di costruire un sistema lineare denso trasferendolo dalla CPU (host) alla GPU (device), preparando i dati necessari per la loro elaborazione in parallelo. Di seguito viene spiegato il funzionamento riga per riga dello pseudocodice.

- Righe 2-3: La variabile `n` viene calcolata come il quadrato della dimensione del reticolo (`size`), corrispondente alla dimensione $n \times n$ della matrice A.
- Riga 4: La matrice A viene costruita sull'host (`h_A`), popolando i suoi elementi con i valori appropriati (ad esempio, 4.0 sulla diagonale principale e -1.0 per i vicini).
- Righe 5-7: La funzione `cudaMemcpy` trasferisce la matrice `h_A` sull'host alla sua controparte `d_A` sul device, e il vettore `h_b` sull'host al vettore `d_b` sul device. Questo trasferimento è essenziale per utilizzare la GPU per i calcoli.
- Righe 8-9: L'handle `cusolverHandle` viene creato tramite la funzione `cusolverDnCreate()`, che permette di configurare e gestire operazioni dense su cuSolver.

Algoritmo 8: Costruzione del sistema lineare per matrici dense

Input:

`h_A, d_A`	Matrice del sistema lineare sull'host e sul device
`h_b, d_b`	Vettore dei termini noti sull'host e sul device
`d_x, h_x`	Vettore soluzione sul device e sull'host
`size`	Dimensione del reticolo ($size \times size$)
`cusolverHandle`	Handle per cuSolver Dense

Output:

Sistema lineare denso costruito su GPU

1 **function** *cuda_build_dense_linear_system(h_A, d_A, h_b, d_b, d_x, h_x, size, cusolverHandle)*
2 **STEP 1:** Inizializza le variabili
3 $n \leftarrow size \times size$
4 **STEP 2:** Popola la matrice *h_A* su CPU
5 **STEP 3:** Copia *h_A* e *h_b* su GPU
6 cudaMemcpy(*d_A*, *h_A*, $n \times n$)
7 cudaMemcpy(*d_b*, *h_b*, n)
8 **STEP 4:** Crea l'handle di cuSolver
9 cusolverDnCreate(*cusolverHandle*)

Funzione cuda_solve_dense_linear_system

La funzione `cuda_solve_dense_linear_system` si occupa della risoluzione di un sistema lineare denso $A \cdot x = b$ utilizzando la libreria `cuSolver Dense`. Questo approccio è adatto per matrici dense ed è stato implementato per confrontare le prestazioni con l'uso di matrici sparse. La funzione segue una sequenza strutturata di passi:

- Riga 2: Inizializza le variabili
  La dimensione del sistema $n = \text{size} \times \text{size}$ e il `leading dimension` (`lda`) vengono calcolati. Vengono allocate le variabili necessarie sulla GPU:
  - `d_info`: Contiene informazioni sull'esito della fattorizzazione.
  - `devIpiv`: Vettore di pivoting utilizzato durante la fattorizzazione LU.
  - `d_work`: Memoria temporanea richiesta per il calcolo.
- Righe 3-6: Calcola il workspace per la fattorizzazione LU
  La funzione `cusolverDnDgetrf_bufferSize` calcola la dimensione del workspace (`lwork`) necessaria per la fattorizzazione LU di una matrice $n \times n$. La memoria viene quindi allocata sulla GPU.
- Righe 7-9: Esegui la fattorizzazione LU
  La funzione `cusolverDnDgetrf` esegue la fattorizzazione LU della matrice A. Questa operazione decompone A in due matrici triangolari, L e U, e utilizza il vettore di pivoting (`devIpiv`) per gestire il riordino delle righe.
- Righe 10-13: Verifica l'esito della fattorizzazione
  Il risultato della fattorizzazione viene copiato dalla GPU alla CPU (`cudaMemcpy`). Se il valore restituito in `info` è diverso da zero, significa che la matrice A è singolare o che si è verificato un errore, e l'esecuzione viene interrotta.
- Righe 14-15: Risolvi il sistema $A \cdot x = b$
  Utilizzando la funzione `cusolverDnDgetrs`, il sistema lineare viene risolto sfruttando la fattorizzazione LU calcolata in precedenza. La soluzione x viene calcolata direttamente sovrascrivendo b.
- Righe 16-17: Copia i risultati sulla CPU
  Il vettore soluzione x viene copiato dalla memoria della GPU (`d_b`) alla memoria dell'host (`h_x`) per ulteriori elaborazioni.
- Righe 18-21: Calcola i campi elettrici
  Il kernel CUDA `calculate_electric_fields` viene lanciato per calcolare le componenti del campo elettrico (E_x e E_y) utilizzando la soluzione x. I risultati vengono quindi copiati dalla GPU alla CPU.
- Righe 22-25: Libera memoria GPU
  La memoria temporanea allocata sulla GPU (`d_info`, `d_work`, `devIpiv`) viene liberata per evitare fughe di memoria.

Al termine della funzione, la soluzione x del sistema lineare è disponibile sulla CPU (`h_x`), insieme ai campi elettrici E_x e E_y. Questi risultati possono essere utilizzati per aggiornare il sistema simulato o per ulteriori analisi.

Algoritmo 9: Risoluzione del sistema lineare per matrici dense

Input:

`d_A, d_b`	Matrice e vettore dei termini noti sul device
`d_x, h_x`	Vettore soluzione sul device e sull'host
`h_E_field_x, d_E_field_x`	Componente x del campo elettrico (host e device)
`h_E_field_y, d_E_field_y`	Componente y del campo elettrico (host e device)
`size`	Dimensione del reticolo ($size \times size$)
`cusolverHandle`	Handle per cuSolver Dense

Output:

Soluzione del sistema lineare denso

function *cuda_solve_dense_linear_system(d_A, d_b, d_x, h_x, h_E_field_x, d_E_field_x, h_E_field_y, d_E_field_y, size, cusolverHandle)*

- **STEP 1:** Inizializza le variabili
- $n \leftarrow size \times size$, $lda \leftarrow n$
- Alloca memoria per d_info, $devIpiv$, d_work
- **STEP 2:** Calcola il workspace per la fattorizzazione LU
- cusolverDnDgetrf_bufferSize($cusolverHandle$, n, n, d_A, lda)
- cudaMalloc(d_work, lwork)
- **STEP 3:** Esegui la fattorizzazione LU
- cusolverDnDgetrf($cusolverHandle$, $n, n, d\_A, lda, d\_work, devIpiv, d\_info$)
- **STEP 4:** Verifica l'esito della fattorizzazione
- cudaMemcpy($info$, d_info)
- **if** $info \neq 0$ **then**
  - Stampa errore e termina
- **STEP 5:** Risolvi il sistema $A \cdot x = b$
- cusolverDnDgetrs($cusolverHandle$, $CUBLAS\_OP\_N, n, 1, d\_A, lda, devIpiv, d\_b, n, d\_info$)
- **STEP 6:** Copia i risultati sulla CPU
- cudaMemcpy(h_x, d_b)
- **STEP 7:** Calcola i campi elettrici
- Lancia kernel `calculate_electric_fields`
- cudaMemcpy($h\_E\_field\_x$, $d\_E\_field\_x$)
- cudaMemcpy($h\_E\_field\_y$, $d\_E\_field\_y$)
- **STEP 8:** Libera memoria GPU
- cudaFree(d_info)
- cudaFree(d_work)
- cudaFree($devIpiv$)

Funzione cuda_build_sparse_linear_system (cuSOLVER e cuSPARSE)

La funzione `cuda_build_sparse_linear_system` è responsabile dell'allocazione e costruzione del sistema lineare sparso $A \cdot x = b$, in cui A è rappresentata in formato CSR (*Compressed Sparse Row*) a differenza del codice C sequenziale, dove la matrice è rappresentata in CSC (*Compressed Sparse Column*). Questa funzione viene eseguita

prima del ciclo di simulazione Monte Carlo per preparare la matrice A e il vettore b necessari per la risoluzione del sistema lineare.

- Riga 2: Calcola dimensioni e inizializza
  La dimensione della matrice A viene calcolata come $n = \text{size} \times \text{size}$, dove `size` è il lato del reticolo. Il numero di elementi non nulli (`nnz`) viene stimato con una funzione apposita (`calculateNNZ`), basandosi sulla struttura del problema.

- Righe 5-8: Crea e popola matrice COO
  La matrice è inizialmente creata in formato COO (*Coordinate List*), che è più semplice da popolare. La funzione `populateSparseMatrixCPU` genera indici di riga, indici di colonna e valori degli elementi non nulli su CPU. Successivamente, i dati vengono copiati dalla memoria host alla GPU per un'elaborazione più efficiente.

- Righe 9-11: Converti COO in CSR
  La matrice `cooMatrix`, allocata in formato COO, viene convertita in formato CSR tramite la funzione `convertCOOtoCSR`. Questo formato è più compatto e adatto per la risoluzione di sistemi lineari su GPU.

- Righe 12-14: Inizializza handle e descrittore
  Vengono creati l'handle di cuSolver (`cusolverHandle`) e il descrittore della matrice CSR (`descrA`). Quest'ultimo viene configurato per specificare il tipo di matrice (generica) e la base degli indici (zero-based).

- Righe 15-17: Libera memoria temporanea
  La matrice COO e le sue variabili temporanee vengono deallocate sia su CPU che su GPU per liberare risorse, lasciando solo la matrice CSR pronta per l'uso.

Algoritmo 10: Costruzione del sistema lineare per matrici sparse (primo approccio)

Input:

`h_chemical`	Campo chimico sull'host
`csrMatrix`	Matrice CSR
`size`	Dimensione del reticolo
`h_b`, `d_b`	Vettori b su host e device
`d_x`, `h_x`	Vettori x su device e host
`cusolverHandle`	Handle di cuSolver
`descrA`	Descrittore della matrice CSR

Output:

`csrMatrix` modificata per riferimento

1 **Function** *CUDA_BUILD_SPARSE_LINEAR_SYSTEM*:
2 **STEP 1:** Calcola dimensioni e inizializza
3 $n \leftarrow size \times size$
4 $nnz \leftarrow$ calculateNNZ$(size)$
5 **STEP 2:** Crea e popola matrice COO
6 $cooMatrix \leftarrow$ createCOOMatrix(n, nnz)
7 Popola *cooMatrix* su CPU
8 Copia *cooMatrix* su GPU
9 **STEP 3:** Converti COO in CSR
10 $csrMatrix \leftarrow$ createCSRMatrix(n, nnz)
11 Converti *cooMatrix* in *csrMatrix*
12 **STEP 4:** Inizializza handle e descrittore
13 $cusolverHandle \leftarrow$ cusolverSpCreate()
14 $descrA \leftarrow$ cusparseCreateMatDescr()
15 Imposta tipo e base indice per $descrA$
16 **STEP 5:** Libera memoria temporanea
17 Libera *cooMatrix* su CPU e GPU

Funzione cuda_solve_sparse_linear_system (cuSOLVER e cuSPARSE)

La funzione `cuda_solve_sparse_linear_system` si occupa della risoluzione del sistema lineare sparso $A \cdot x = b$ utilizzando la libreria `cuSolver`. Oltre a calcolare la soluzione x, questa funzione utilizza il risultato per determinare i campi elettrici associati.

- Righe 2-3: Copia b su GPU
  Il vettore b, contenente i valori del campo chimico sull'host (`h_chemical`), viene copiato nella memoria della GPU (`d_b`) per l'elaborazione.
- Righe 4-12: Risolvi $Ax = b$
  La funzione `cusolverSpDcsrlsvqr` viene utilizzata per risolvere il sistema lineare. Questa funzione esegue una fattorizzazione QR della matrice CSR `A`. Il parametro `tol` definisce la tolleranza per il calcolo, mentre `reorder` consente un riordino degli indici per migliorare l'efficienza. Se la matrice è singolare, viene indicata la posizione del problema e stampato un errore. In caso contrario, la soluzione viene calcolata correttamente e stampata.

- Righe 13-17: Calcola campi elettrici
  Utilizzando la soluzione x, viene lanciato il kernel `calculate_electric_fields` sulla GPU. Questo kernel calcola i campi elettrici nei due assi (E_x e E_y), che vengono successivamente copiati nella memoria host (`h_E_field_x` e `h_E_field_y`).

Al termine della funzione, i campi elettrici calcolati sono disponibili sull'host (`h_E_field_x` e `h_E_field_y`), pronti per ulteriori elaborazioni o analisi.

Algoritmo 11: Risoluzione del sistema lineare per matrici sparse (primo approccio)

Input:

`csrMatrix`: Matrice CSR	
`d_b`, `h_b`	Vettori b su device e host
`d_x`, `h_x`	Vettori x su device e host
`h_chemical`	Campo chimico sull'host
`h_E_field_x`, `d_E_field_x`	Campo elettrico x su host e device
`h_E_field_y`, `d_E_field_y`	Campo elettrico y su host e device
`size`	Dimensione del reticolo
`cusolverHandle`	Handle di cuSolver
`descrA`	Descrittore della matrice CSR

Output:

`h_E_field_x`, `h_E_field_y`	Campi elettrici calcolati

```
1  Function cuda_solve_sparse_linear_system:
2      STEP 1: Copia b su GPU
3      Copia h_chemical → d_b
4      STEP 2: Risolvi Ax = b
5      tol ← 1e−6
6      reorder ← 1
7      singularity ← −1
8      Risolvi con cusolverSpDcsrlsvqr()
9      if singularity ≥ 0 then
10         Stampa errore matrice singolare
11     else
12         Stampa soluzione corretta
13     STEP 3: Calcola campi elettrici
14     Configura griglia e blocchi CUDA
15     Lancia kernel calculate_electric_fields()
16     Copia d_E_field_x → h_E_field_x
17     Copia d_E_field_y → h_E_field_y
```

Funzione cuda_build_sparse_linear_system_cuDSS

- Righe 2-4: Calcola dimensioni e inizializza
  La dimensione della matrice è calcolata come il quadrato della dimensione del reticolo ($n = \text{size} \times \text{size}$). Inoltre, viene stimato il numero di elementi non nulli (`nnz`) nella matrice per ottimizzare l'allocazione della memoria.

- Righe 5-8: Crea e popola matrice COO
  Si inizializza una matrice in formato COO (Coordinate List), utile per popolare

i valori e gli indici delle matrici sparse. Successivamente, i dati della matrice vengono trasferiti dalla CPU alla GPU.

- Righe 9-11: Converti COO in CSR
  La matrice in formato COO viene convertita in formato CSR (Compressed Sparse Row), più efficiente per le operazioni di calcolo e supportato direttamente da cuDSS.

- Righe 12-18: Inizializza handle e configurazione cuDSS
  Si crea un handle per cuDSS (`cudssHandle`) e uno stream CUDA per la parallelizzazione asincrona. Si configurano il solver (`solverConfig`) e i dati necessari (`solverData`) per la risoluzione del sistema.

- Righe 19-22: Crea matrici dense e sparse
  Le matrici del sistema (sparse per A, dense per b e x) vengono create utilizzando i descrittori appropriati per essere gestite da cuDSS.

- Righe 23-24: Fattorizzazione simbolica
  Si esegue l'analisi simbolica del sistema lineare, necessaria per preparare la matrice alla fattorizzazione numerica e alla successiva risoluzione.

- Righe 25-26: Libera memoria temporanea
  Le strutture di supporto, come la matrice COO, vengono liberate per ottimizzare l'utilizzo della memoria e ridurre i costi computazionali.

Algoritmo 12: Costruzione del sistema lineare per matrici sparse (implementazione definitiva)

Input:

`h_chemical`	Campo chimico sull'host
`csrMatrix`	Matrice CSR
`size`	Dimensione del reticolo
`h_b`, `d_b`	Vettori b su host e device
`d_x`, `h_x`	Vettori x su device e host
`cudssHandle`	Handle di cuDSS
`solverConfig`, `solverData`	Configurazione e dati del solver
`x`, `b`, `A`	Matrici dense e sparsa in formato cuDSS

Output:

`csrMatrix` modificata per riferimento

1 **Function** *cuda_build_sparse_linear_system_cuDSS*:
2 **STEP 1:** Calcola dimensioni e inizializza
3 $n \leftarrow size \times size$
4 $nnz \leftarrow$ calculateNNZ$(size)$
5 **STEP 2:** Crea e popola matrice COO
6 $cooMatrix \leftarrow$ createCOOMatrix(n, nnz)
7 Popola $cooMatrix$ su CPU
8 Copia $cooMatrix$ su GPU
9 **STEP 3:** Converti COO in CSR
10 $csrMatrix \leftarrow$ createCSRMatrix(n, nnz)
11 Converti $cooMatrix$ in $csrMatrix$
12 **STEP 4:** Inizializza handle e configurazione cuDSS
13 $cudssHandle \leftarrow$ cudssCreate()
14 $stream \leftarrow$ createCUDAStream()
15 Imposta $stream$ nell'handle cuDSS
16 $solverConfig \leftarrow$ cudssConfigCreate()
17 $solverData \leftarrow$ cudssDataCreate$(cudssHandle)$
18 **STEP 5:** Crea matrici dense e sparse
19 $b \leftarrow$ cudssMatrixCreateDn$(ncols, nrhs, ldb, d_b)$
20 $x \leftarrow$ cudssMatrixCreateDn$(nrows, nrhs, ldx, d_x)$
21 $A \leftarrow$ cudssMatrixCreateCsr$(csrMatrix)$
22 **STEP 6:** Fattorizzazione simbolica
23 cudssExecute(cudssHandle, PHASE_ANALYSIS, solverConfig, solverData, A, x, b)
24 **STEP 7:** Libera memoria temporanea
25 Libera $cooMatrix$ su CPU e GPU

Funzione cuda_solve_sparse_linear_system_cuDSS

- Righe 2-3: Aggiorna il lato destro
  I dati di input (`h_chemical`) vengono copiati dal lato host al lato device (`d_b`) per consentire l'elaborazione parallela sulla GPU.

- Riga 4: Fattorizzazione numerica
  La matrice sparsa viene fattorizzata numericamente utilizzando le informazio-

ni pre-calcolate nella fase simbolica. Questo passaggio prepara il sistema alla risoluzione.

- Righe 5-6: Risoluzione del sistema
  La soluzione del sistema lineare viene calcolata utilizzando la matrice fattorizzata e il vettore b, restituendo il risultato come vettore x.
- Riga 7: Sincronizzazione
  La sincronizzazione del dispositivo CUDA assicura che tutte le operazioni siano completate prima di accedere ai risultati.
- Riga 8: Restituisci la soluzione
  Il vettore soluzione (`d_x`) viene restituito come risultato, pronto per essere utilizzato in ulteriori calcoli o analisi.

Algoritmo 13: Risoluzione del sistema lineare per matrici sparse (implementazione definitiva)

Input:

`csrMatrix`	Matrice CSR già popolata
`d_b`	Vettore b sul device
`h_chemical`	Campo chimico sull'host
`d_x`	Vettore x sul device
`size`	Dimensione del reticolo
`cudssHandle`	Handle di cuDSS
`solverConfig`, `solverData`	Configurazione e dati del solver
`x`, `b`, `A`	Matrici dense e sparsa in formato cuDSS

Output:

`d_x`	Soluzione del sistema lineare

1 **Function** *CUDA_SOLVE_SPARSE_LINEAR_SYSTEM_CUDSS*:
2 **STEP 1:** Aggiorna il lato destro
3 Copia *h_chemical* in *d_b* sul device
4 **STEP 2:** Fattorizzazione numerica
5 cudssExecute(cudssHandle, PHASE_FACTORIZATION, solverConfig, solverData, A, x, b)
6 **STEP 3:** Risoluzione del sistema
7 cudssExecute(cudssHandle, PHASE_SOLVE, solverConfig, solverData, A, x, b)
8 **STEP 4:** Sincronizzazione
9 Sincronizza il dispositivo CUDA
10 **STEP 5:** Restituisci la soluzione
11 Return *d_x*

4.1.5 Aggiornamento delle cellule del reticolo

L'aggiornamento delle cellule del reticolo è una fase essenziale del modello, in cui i valori chimici vengono modificati per riflettere la diffusione, il decadimento e la secrezione di sostanze chimiche da parte delle cellule. Questa operazione richiede il calcolo del laplaciano su una griglia bidimensionale e l'aggiornamento dei valori

in base ai parametri fisici definiti, come il tasso di diffusione, il decadimento e la secrezione.

Per ottimizzare questa operazione su reticoli di grandi dimensioni, l'implementazione è stata suddivisa in due funzioni principali:

- `update_chemical_kernel`: un kernel CUDA che esegue i calcoli richiesti per ogni cella del reticolo in parallelo, sfruttando la capacità della GPU di elaborare simultaneamente molteplici elementi.

- `update_chemical_cuda`: una funzione host che coordina l'intero processo di aggiornamento, inclusi il lancio del kernel CUDA, la sincronizzazione e il trasferimento dei dati tra l'host e il device.

Questa suddivisione consente di massimizzare l'efficienza computazionale e mantenere una chiara separazione tra i calcoli paralleli e la gestione dei dati. Nel prossimo paragrafo, spiegheremo in dettaglio il funzionamento di ciascuna funzione.

Funzione update_chemical_kernel

La funzione `update_chemical_kernel` è un kernel CUDA che aggiorna il campo chimico su ogni cella del reticolo. Utilizza la diffusione e altri parametri fisici per calcolare il nuovo stato chimico, eseguendo i calcoli in parallelo per tutte le celle.

- Righe 2-3: Calcola l'indice globale del thread
  L'indice globale del thread (`idx`) viene calcolato come combinazione dell'indice del blocco (`blockIdx.x`) e del thread all'interno del blocco (`threadIdx.x`). Questo indice è univoco e determina quale elemento del reticolo viene elaborato dal thread.

- Righe 4-5: Calcola le coordinate bidimensionali
  Utilizzando `idx`, si calcolano le coordinate bidimensionali (`x` e `y`) della cella corrispondente nel reticolo. Questi valori sono fondamentali per accedere correttamente agli elementi della matrice rappresentata in formato 1D.

- Riga 7: Verifica la validità delle coordinate
  Le coordinate calcolate (`x < size` e `y < size`) vengono verificate per assicurarsi che il thread non acceda a memoria non valida, specialmente nei casi in cui il numero di thread superi il numero di celle.

- Riga 8: Inizia il calcolo del laplaciano
  Il calcolo del laplaciano inizia assegnando a `laplacian` il contributo centrale come `-4 * prev_chemical[x * size + y]`.

- Righe 9-15: Itera sui vicini della cella
  Si utilizzano due cicli annidati (`i` e `j`) per iterare su tutte le posizioni circostanti della cella corrente, incluse le diagonali. La condizione `if (i == 0 && j == 0)` assicura che la cella corrente non contribuisca due volte al calcolo del laplaciano. Le coordinate dei vicini (`xx` e `yy`) vengono calcolate tenendo conto del bordo del reticolo tramite l'operazione modulo (%), garantendo condizioni al contorno periodiche.

- Riga 15: Aggiorna il laplaciano
  Il valore chimico del vicino (`prev_chemical[xx * size + yy]`) viene aggiunto al laplaciano.
- Righe 16-20: Aggiorna il valore chimico corrente
  Il valore chimico corrente (`curr_chemical[x * size + y]`) viene aggiornato applicando l'equazione di diffusione, che include:
  - Il contributo della diffusione: `DIFFUSION * laplacian`.
  - Il decadimento proporzionale al valore chimico corrente: `-DECAY_RATE * prev_chemical[x * size + y]`.
  - La secrezione, che è attiva solo per le celle occupate: `SECRETION_RATE * (lattice[x * size + y] > 0 ? 1 : 0)`.

Motivazioni per la parallelizzazione

- Ogni cella del reticolo può essere aggiornata indipendentemente dalle altre, rendendo questa operazione ideale per il calcolo parallelo.
- La GPU permette di elaborare contemporaneamente un gran numero di celle, riducendo i tempi di calcolo rispetto a un approccio sequenziale.
- L'uso delle condizioni al contorno periodiche sfrutta il modulo per semplificare la gestione dei vicini ai bordi del reticolo.

Algoritmo 14: Aggiorna il campo chimico su ogni cella del reticolo (kernel)

Input:

`prev_chemical`	Matrice chimica dello stato precedente sul device
`curr_chemical`	Matrice chimica dello stato corrente sul device
`lattice`	Matrice del reticolo sul device
`size`	Dimensione del reticolo

Output:

`curr_chemical`	Matrice aggiornata con il nuovo stato chimico

```
1  kernel update_chemical_kernel(prev_chemical, curr_chemical, lattice, size):
2      STEP 1: Calcola gli indici
3      idx ← blockIdx.x × blockDim.x + threadIdx.x
4      x ← idx/size                                              Coordinata x
5      y ← idx mod size                                          Coordinata y
6      STEP 2: Verifica validità dell'indice
7      if x < size and y < size then
8          STEP 3: Calcola il laplaciano
9          laplacian ← −4 × prev_chemical[x × size + y]
10         for i ← −1 to 1 do
11             for j ← −1 to 1 do
12                 if i ≠ 0 or j ≠ 0 then
13                     xx ← (x + i + size) mod size
14                     yy ← (y + j + size) mod size
15                     laplacian ← laplacian + prev_chemical[xx × size + yy]
16         STEP 4: Aggiorna il valore chimico
17         curr_chemical[x × size + y] ← prev_chemical[x × size + y] +
18                 DIFF_DT × (DIFFUSION × laplacian −
19                 DECAY_RATE × prev_chemical[x × size + y] +
20                 SECRETION_RATE × (lattice[x × size + y] > 0?1 : 0))
```

Funzione update_chemical_cuda

La funzione `update_chemical_cuda` è una funzione host che coordina l'intero processo di aggiornamento del campo chimico utilizzando la GPU. Questa funzione include il trasferimento dei dati tra l'host e il device, il lancio del kernel CUDA per eseguire i calcoli paralleli e la sincronizzazione dei dati aggiornati.

- Righe 2-5: Configura i parametri CUDA
  Configura il numero di thread per blocco (`threads_per_block`) come 256, un valore standard che bilancia l'efficienza computazionale e l'occupazione della GPU. Calcola il numero totale di celle nel reticolo come `size²`, poiché il reticolo bidimensionale è rappresentato come un array unidimensionale. Determina il numero di blocchi necessari (`blocks_per_grid`) per coprire tutte le celle del reticolo, dividendo il numero totale di celle per il numero di thread per blocco e arrotondando per eccesso.

- Righe 6-8: Copia i dati dall'host al device
  Copia i dati del reticolo (`h_lattice`) dalla memoria host alla memoria del de-

vice (d_lattice) utilizzando cudaMemcpy. Questo trasferimento è necessario per garantire che i dati del reticolo siano accessibili al kernel CUDA. Successivamente, copia i dati chimici iniziali (h_curr_chemical) dall'host al device (d_curr_chemical) per preparare la GPU ad aggiornare il campo chimico.

- Righe 9-13: Ciclo di aggiornamento chimico
  Esegue un ciclo di 10 iterazioni per simulare l'evoluzione temporale del campo chimico. A ogni iterazione:
  - Lancia il kernel CUDA update_chemical_kernel con blocks_per_grid blocchi e threads_per_block thread per blocco. Il kernel aggiorna il campo chimico per ogni cella del reticolo.
  - Sincronizza il dispositivo con cudaDeviceSynchronize per assicurarsi che tutti i calcoli siano completati prima di procedere.
  - Esegue uno scambio (swap) tra d_prev_chemical e d_curr_chemical (i due puntatori delle matrici chimiche) per preparare il dispositivo alla prossima iterazione.
- Riga 14-15: Copia i dati dal device all'host
  Copia i dati chimici aggiornati da d_prev_chemical a h_curr_chemical (dal device all'host) per renderli disponibili al programma host. Questo trasferimento finale garantisce che i risultati calcolati sulla GPU siano accessibili per ulteriori elaborazioni o analisi.

Motivazioni per la suddivisione in fasi

La funzione update_chemical_cuda è stata progettata per suddividere chiaramente le operazioni di trasferimento dei dati, calcolo parallelo e sincronizzazione:

- La configurazione e il trasferimento iniziali (Righe 1-5) preparano la GPU a eseguire calcoli paralleli, riducendo l'overhead durante l'esecuzione del kernel.
- L'uso di un ciclo per le iterazioni temporali consente di simulare l'evoluzione dinamica del campo chimico.
- La separazione tra kernel CUDA e funzione host semplifica la gestione della memoria e la sincronizzazione dei dati tra host e device.

Vantaggi dell'approccio

- Il calcolo parallelo con CUDA riduce significativamente i tempi di esecuzione rispetto a un approccio sequenziale, specialmente per reticoli di grandi dimensioni.
- La modularità del codice rende più semplice il debugging e facilita eventuali miglioramenti futuri.
- Il processo iterativo permette di catturare l'evoluzione temporale del sistema, mantenendo al contempo un'alta efficienza computazionale.

Algoritmo 15: update_chemical_cuda

Input:

`d_prev_chemical`	Matrice chimica stato precedente (device)
`h_curr_chemical`	Matrice chimica stato corrente (host)
`d_curr_chemical`	Matrice chimica stato corrente (device)
`d_lattice`	Matrice del reticolo (device)
`h_lattice`	Matrice del reticolo (host)
`size`	Dimensione del reticolo

Output:

`h_curr_chemical`	Matrice chimica aggiornata (host)

```
1  function update_chemical_cuda(d_prev_chemical, h_curr_chemical,
   d_curr_chemical, d_lattice, h_lattice, size)                    :
2      STEP 1: Configura i parametri CUDA
3      threads_per_block ← 256
4      total_cells ← size^2
5      blocks_per_grid ← ⌈total_cells/threads_per_block⌉
6      STEP 2: Copia i dati dall'host al device
7      cudaMemcpy(d_lattice, h_lattice)
8      cudaMemcpy(d_curr_chemical, h_curr_chemical)
9      STEP 3: Ciclo di aggiornamento chimico
10     for t ← 1 to 10 do
11         update_chemical_kernel<<< blocks_per_grid,
           threads_per_block >>>(d_prev_chemical, d_curr_chemical, d_lattice, size)
12         Synchronize device
13         Swap d_prev_chemical and d_curr_chemical
14     STEP 4: Copia i dati dal device all'host
15     cudaMemcpy(h_curr_chemical, d_prev_chemical)
```

4.1.6 Normalizzazione del campo chimico

La normalizzazione del campo chimico è necessaria per garantire che i valori chimici rimangano scalati correttamente dopo ogni passo Monte Carlo. Per ottimizzare questa operazione su reticoli di grandi dimensioni, è stata implementata una strategia parallela, suddivisa in due fasi principali: il calcolo del massimo valore chimico e la normalizzazione vera e propria.

L'integrazione di CUB (*CUDA UnBound*) nel progetto ha permesso di migliorare significativamente l'efficienza delle routine di calcolo del massimo e di normalizzazione, sostituendo le precedenti implementazioni personalizzate con primitive ottimizzate per GPU. In particolare, l'utilizzo della funzione `cub::DeviceReduce::Max` ha consentito di calcolare il massimo degli elementi di un array direttamente sulla GPU, sfruttando una riduzione parallela altamente performante. Questo approccio ha ridotto l'overhead computazionale, garantendo al contempo un accesso coalescente alla memoria globale.

Per la normalizzazione, la combinazione di CUB con un semplice kernel CUDA ha semplificato l'applicazione dell'operazione di divisione elemento per elemento, mantenendo elevate le prestazioni. L'uso di un kernel CUDA per la normaliz-

zazione consente di sfruttare direttamente il valore massimo calcolato dalla funzione `cub::DeviceReduce::Max`, eliminando la necessità di trasferimenti ripetuti di dati tra CPU e GPU.

Grazie a queste migliorie, il tempo di esecuzione delle routine chiave del progetto è stato notevolmente ridotto, portando a un aumento complessivo delle prestazioni del sistema. L'integrazione di CUB si è dimostrata una soluzione efficace e scalabile, particolarmente utile per sfruttare al meglio le capacità di parallelismo delle GPU NVIDIA nelle operazioni di pre-elaborazione e analisi dei dati.

CUB: CUDA UnBound

NVIDIA include nel suo toolkit di sviluppo CUDA la libreria CUB, una collezione di algoritmi paralleli ottimizzati per eseguire operazioni ad alte prestazioni su GPU [8] (*NVIDIA Corporation*, *2023*). CUB è progettata per fornire primitive ad alte prestazioni per operazioni di riduzione, scansione, segmentazione, trasformazione e altro, sfruttando al massimo le capacità di calcolo parallelo delle GPU di NVIDIA.

CUB è una libreria flessibile, altamente scalabile e compatibile con un'ampia gamma di dispositivi NVIDIA, dalle architetture Maxwell fino alle più recenti Hopper. Grazie alla sua struttura modulare e al supporto per un'ampia varietà di tipi di dati, CUB si integra facilmente in pipeline di calcolo GPU, fornendo un'alternativa più efficiente e ottimizzata rispetto alle implementazioni manuali.

Uno dei principali punti di forza di CUB è l'ottimizzazione per le operazioni comuni, come riduzioni e trasformazioni, che garantiscono una minimizzazione dell'overhead computazionale e un accesso coalescente alla memoria [9] (*NVIDIA Corporation*, *2023*). Questo rende CUB una libreria ideale per applicazioni che richiedono alte prestazioni in presenza di grandi volumi di dati, come simulazioni fisiche, analisi di dati scientifici e apprendimento automatico.

L'integrazione di CUB nel progetto ha permesso di migliorare significativamente le prestazioni, sostituendo implementazioni custom con funzioni già ottimizzate. In particolare, l'utilizzo di primitive come `DeviceReduce::Max` e `DeviceScan` ha reso possibile eseguire operazioni di riduzione e normalizzazione in parallelo, con un aumento della velocità computazionale rispetto alle soluzioni precedenti.

All'interno del progetto, l'integrazione di CUB ha richiesto l'utilizzo delle seguenti funzioni principali:

- **DeviceReduce::Max**: consente di calcolare il massimo di un array in parallelo, utilizzando una riduzione ottimizzata.
- **DeviceTransform**: permette di applicare trasformazioni elemento per elemento direttamente sulla GPU, utilizzando iteratori personalizzati.
- **DeviceScan**: implementa scansioni parallele, ideali per la somma cumulativa e altre operazioni di pre-elaborazione dei dati.

Un aspetto distintivo di CUB è l'uso di buffer temporanei personalizzabili, che permettono di ottimizzare l'allocazione della memoria durante l'esecuzione delle operazioni. Grazie a questa flessibilità, gli sviluppatori possono gestire in modo più efficiente le risorse della GPU, riducendo l'overhead delle chiamate e aumentando l'occupazione del dispositivo.

L'integrazione di CUB nel progetto ha portato a un notevole incremento delle prestazioni, riducendo il tempo di esecuzione delle operazioni chiave e garantendo un utilizzo più efficiente della GPU. Grazie a queste migliorie, CUB si è dimostrata una scelta eccellente per ottimizzare il flusso di calcolo parallelo del progetto.

Funzione find_max_CUB

La funzione `find_max_CUB` calcola il massimo valore chimico all'interno del reticolo utilizzando la primitiva di riduzione `cub::DeviceReduce::Max`. Questa funzione alloca la memoria temporanea necessaria, esegue la riduzione parallela e restituisce il massimo valore trovato.

- Righe 2-3: Inizializza il buffer temporaneo
  Inizializza il puntatore `d_temp_storage` come nullo ed imposta a 0 la variabile `temp_storage_bytes`. Questo passo serve per determinare la dimensione della memoria necessaria per l'operazione di riduzione.
- Riga 5: Calcola la dimensione della memoria temporanea
  Effettua una chiamata preliminare alla funzione `cub::DeviceReduce::Max`, che non esegue ancora la riduzione, ma calcola la quantità di memoria necessaria per il buffer temporaneo.
- Righe 6-7: Alloca memoria sulla GPU
  Alloca memoria sulla GPU per il buffer temporaneo (`d_temp_storage`) utilizzando la dimensione calcolata in precedenza. Inoltre, alloca memoria per il valore massimo (`d_max`) che verrà calcolato.
- Riga 10: Calcola il massimo
  Esegue la chiamata effettiva a `cub::DeviceReduce::Max`, che calcola il valore massimo nell'array `d_array` in modo parallelo e salva il risultato in `d_max`.
- Riga 12: Copia il risultato sulla CPU
  Copia il valore massimo calcolato dalla memoria della GPU (`d_max`) alla memoria della CPU (`h_max`) per consentire un utilizzo successivo del risultato.
- Righe 14-15: Libera la memoria sulla GPU
  Libera la memoria allocata per il buffer temporaneo (`d_temp_storage`) e per il valore massimo (`d_max`), completando così la gestione delle risorse sulla GPU.

Algoritmo 16: Calcola il massimo valore chimico (normalizzazione del campo chimico)

Input:
`d_array` — Array sul device
`size` — Dimensione del reticolo
Output:
`h_max` — Valore massimo

1 **Function** *find_max_CUB(d_array, size)*:
2 `d_temp_storage` ← `nullptr`
3 `temp_storage_bytes` ← 0
4 **STEP 1**: Calcola la dimensione della memoria temporanea necessaria
5 `cub::DeviceReduce::Max(d_temp_storage, temp_storage_bytes, d_array)`
6 **STEP 2:** Alloca la memoria sulla GPU
7 `cudaMalloc(d_temp_storage)` — Memoria temporanea
8 `cudaMalloc(d_max)` — Per memorizzare il risultato
9 **STEP 3**: Riduzione per trovare il massimo
10 `cub::DeviceReduce::Max(d_temp_storage, temp_storage_bytes, d_array, d_max)`
11 **STEP 4**: Copia il risultato dalla GPU alla CPU
12 Copia `d_max` in `h_max` sull'host
13 **STEP 5**: Libera la memoria sulla GPU
14 cudaFree(`d_temp_storage`)
15 cudaFree(`d_max`)

Funzione normalize_kernel

La funzione `normalize_kernel` è una funzione kernel che normalizza gli elementi di un array dividendo ciascun elemento per un valore massimo fornito. Ogni thread della GPU elabora un elemento specifico dell'array.

- Riga 2: Calcolo dell'indice globale del thread
  Ogni thread della GPU calcola il proprio indice globale (`idx`) combinando l'indice del blocco e l'indice del thread all'interno del blocco (`blockIdx.x` e `threadIdx.x`). Questo valore viene usato per identificare univocamente l'elemento dell'array da elaborare.

- Riga 3: Controllo della validità dell'indice
  La condizione `if (idx < size)` verifica che l'indice globale del thread sia valido, ovvero all'interno dei limiti dell'array `d_array`. Questa verifica evita accessi non validi alla memoria, che potrebbero causare errori di esecuzione.

- Riga 4: Normalizzazione dell'elemento dell'array
  Se la condizione precedente è soddisfatta, il thread normalizza il valore dell'elemento dell'array `d_array[idx]` dividendo il suo valore corrente per il massimo (`max_value`). Questa operazione garantisce che tutti i valori nell'array siano scalati rispetto al valore massimo, risultando in un array normalizzato.

Algoritmo 17: Normalizza gli elementi di un array (kernel)

Input:

`d_array`	Array sul device
`max_value`	Valore massimo
`size`	Dimensione array

Output:

`d_array`	Array normalizzato

```
1 kernel normalize_kernel(d_array, max_value, size):
2     idx ← Global thread index
3     if idx < size then
4         d_array[idx] ← d_array[idx] / max_value
```

Funzione normalize_CUB

La funzione `normalize_CUB` è una funzione host che gestisce la normalizzazione dell'array lanciando il kernel `normalize_kernel` con una configurazione ottimale di blocchi e thread. Sincronizza la GPU per garantire che tutte le operazioni siano completate.

- Riga 2: Configura il numero di thread per blocco
  Imposta `threads_per_block` a 256, un valore standard per ottimizzare l'occupazione della GPU e garantire un'esecuzione parallela efficiente.
- Riga 3: Calcola il numero di blocchi per griglia
  Calcola `blocks_per_grid` come il numero di blocchi necessari per coprire l'intero array `d_array`. Questo valore è determinato dividendo la dimensione totale dell'array (`size`) per il numero di thread per blocco (`threads_per_block`) e arrotondando per eccesso.
- Riga 5: Lancia il kernel di normalizzazione
  Usa la configurazione di blocchi e thread per lanciare il kernel `normalize_kernel` sulla GPU. Ogni thread normalizza un elemento dell'array `d_array`, dividendo il valore dell'elemento per `max_value`.
- Riga 7: Sincronizza la GPU
  Chiama `cudaDeviceSynchronize()` per sincronizzare la GPU con la CPU. Questo garantisce che tutte le operazioni di normalizzazione avviate sulla GPU siano completate prima che il programma continui l'esecuzione.

Algoritmo 18: Normalizza gli elementi di un array (host)

Input:

`d_array`	Device array
`max_value`	Maximum value
`size`	Dimensione dell'array

Output:

`d_array`	Array normalizzato

```
1 Function normalize_CUB(d_array, max_value, size):
2     threads_per_block ← 256 ;
3     blocks_per_grid ← ⌈size / threads_per_block⌉ ;
4     STEP 1: Lancia la funzione kernel per normalizzare l'array
5     normalize_kernel<<<blocks_per_grid,
       threads_per_block>>>(d_array, max_value, size) ;
6     STEP 2: Sincronizza la GPU
7     cudaDeviceSynchronize() ;
8 end
```

Funzione `normalize_array_CUB`

Esegue l'intero processo di normalizzazione: prima calcola il massimo valore dell'array utilizzando `find_max_CUB`, quindi normalizza l'array chiamando `normalize_CUB`. Combina le operazioni di calcolo del massimo e di normalizzazione in una funzione unica.

- Riga 3: Trova il massimo valore usando CUB
  Chiama la funzione `find_max_with_cub`, che utilizza la funzione primitiva `cub::DeviceReduce::Max` per calcolare il massimo valore nell'array `d_array` direttamente sulla GPU. Il massimo calcolato viene memorizzato nella variabile `max_value`.

- Riga 5: Verifica il valore massimo
  Controlla che il valore massimo calcolato (`max_value`) sia maggiore di zero, per evitare divisioni per zero nella successiva fase di normalizzazione.

- Riga 6: Normalizza l'array usando il kernel CUDA
  Se il massimo è valido (`max_value > 0`), chiama `normalize_with_cub`, che lancia il kernel CUDA per normalizzare ciascun elemento dell'array `d_array`, dividendo il suo valore per `max_value`.

Algoritmo 19: Normalizzazione del campo chimico

Input:
`d_array` — Array sul device
`size` — Dimensione array
Output: `d_array` — Array normalizzato

```
1 Function normalize_array_CUB:
2     STEP 1: Trova il valore massimo usando CUB
3     max_value ← find_max_with_cub(d_array, size)
4     STEP 2: Normalizza l'array usando il kernel CUDA
5     if max_value > 0 then
6         normalize_with_cub(d_array, max_value, size)
```

4.1.7 Risultati e confronto

In questa sezione vengono analizzati i risultati ottenuti dal codice parallelo e messi a confronto con le versioni sequenziali sviluppate in MATLAB e C. I test sono stati eseguiti sulla stessa macchina utilizzata per il confronto dei codici sequenziali, come descritto in precedenza. Sebbene l'hardware della macchina presenti caratteristiche tecniche ormai datate, l'uso della GPU ha permesso di ottenere significativi miglioramenti in termini di efficienza rispetto alle implementazioni sequenziali.

Grazie all'elaborazione parallela sulla GPU, è stato possibile eseguire simulazioni su reticoli di dimensioni molto più grandi rispetto a quelle gestibili dalle versioni sequenziali. Questo ha permesso di valutare non solo le prestazioni, ma anche la scalabilità e l'efficacia del codice parallelo su problemi di larga scala.

Le specifiche delle macchine utilizzate sono le seguenti:

- **Lenovo ideapad Y700-15ISK Laptop (2015)**

- Processore: Intel(R) Core i7-6700HQ, 2.60 GHz, 4 Core

- RAM: 16 GB, DDR4-2133 MHz

- GPU: NVIDIA GeForce GTX 960M

- VRAM: 4GB

Prestazioni del codice parallelo

Il codice parallelo sviluppato rappresenta un punto di svolta in termini di prestazioni rispetto alle versioni sequenziali e MATLAB. Sfruttando l'architettura delle GPU NVIDIA, le operazioni di calcolo del massimo e di normalizzazione sono state ottimizzate utilizzando la libreria CUB, mentre la risoluzione del sistema lineare è stata significativamente accelerata grazie all'integrazione della libreria cuDSS.

Confrontando i risultati:

- Il codice parallelo è di un ordine di grandezza più veloce rispetto alla versione sequenziale per tutte le operazioni principali, tra cui calcolo del massimo, normalizzazione e risoluzione del sistema lineare.

- Rispetto al codice MATLAB, il miglioramento è ancora più evidente: il codice parallelo supera ampiamente le prestazioni della versione MATLAB, dimostrando che l'ottimizzazione su GPU non solo accelera i calcoli ma li rende scalabili a dataset di dimensioni molto maggiori.

L'integrazione di cuDSS ha permesso di sfruttare una libreria progettata specificamente per risolvere sistemi lineari con matrici sparse in modo diretto e ottimizzato per GPU. In particolare, l'uso di funzioni come `cudssExecute()` ha consentito di ridurre drasticamente i tempi di risoluzione rispetto alle precedenti implementazioni basate su cuSOLVER o metodi CPU. Grazie alla scalabilità di
cuDSS, il sistema può affrontare problemi di grandi dimensioni con un utilizzo ottimale delle risorse GPU.

Per quanto riguarda le operazioni di calcolo del massimo e normalizzazione, la combinazione tra CUB e kernel CUDA personalizzati ha garantito un'elevata efficienza in tutte le fasi del processo, eliminando i colli di bottiglia e sfruttando al meglio il parallelismo massivo della GPU.

L'uso congiunto di CUB per l'ottimizzazione delle operazioni di riduzione, di cuDSS per la risoluzione del sistema lineare e di routine fatte *ad hoc* per altre parti parallelizzate come per il calcolo dei campi elettrici e l'inizializzazione delle cellule del reticolo, ha permesso al codice parallelo di raggiungere prestazioni senza precedenti, rendendolo la soluzione ideale per applicazioni computazionalmente intensive su dataset di grandi dimensioni.

I parametri di simulazione utilizzati per la prima e la seconda simulazione coincidono con quelli impiegati nell'analisi dei risultati della parte sequenziale, riportati nelle Tabelle 2.1, 2.2, 2.3 nel capitolo dedicato. Per confrontare i tempi di simulazione per un reticolo di dimensioni standard 200x200 tra MATLAB, C e CUDA sulla stessa macchina:

Tabella 4.1: Confronto di tempi di simulazione tra MATLAB, C e CUDA sul reticolo 200x200, 10 cellule e 100 MCS

MATLAB	**C**	**CUDA**
00:06:42.355	00:01:04.600	00:00:06.243

Dalla Tabella 4.1 emerge chiaramente che, anche per un reticolo di dimensioni ridotte, la versione parallela in CUDA risulta essere la più efficiente in termini di tempo di esecuzione. Ciononostante, per comprendere appieno il comportamento del codice CUDA e sfruttare al massimo le potenzialità della parallelizzazione, è opportuno eseguire l'analisi su larga scala, al fine di poter apprezzare ancor di più la differenza in termini di esecuzione tra i vari programmi proposti.

Confronto per moduli

Per fornire un'analisi più dettagliata, sono stati misurati i tempi di esecuzione di ogni modulo del codice. Questo confronto consente di valutare l'efficienza delle funzioni implementate in parallelo rispetto alle loro controparti sequenziali. I parametri di simulazione sono gli stessi usati per il confronto tra MATLAB e C, presenti nella Tabella 2.1.

Tabella 4.2: Confronto dei tempi di esecuzione per la simulazione 1

Modulo	**C**	**CUDA**
Inizializza cellule	0.000234 s	0.000455 s
Risoluzione sistema lineare	52.399255 s	4.758289 s
Calcolo campi elettrici	0.038136 s	0.000860 s
Aggiorna campo chimico	2.617650 s	0.697437 s
Normalizza campo chimico	0.019288 s	0.018267 s
Tempo MCS complessivo	55.911373 s	6.114147 s

Reticolo: 200x200, MCS: 100, Numero Cellule: 10.
Matrice A: 1600000000, Vettore b: 40000.

Tabella 4.3: Confronto dei tempi di esecuzione per la simulazione 2

Modulo	**C**	**CUDA**
Inizializza cellule	0.000986 s	0.000877 s
Risoluzione sistema lineare	943.357168 s	37.290616 s
Calcolo campi elettrici	0.315063 s	0.002663 s
Aggiorna campo chimico	20.691565 s	5.379040 s
Normalizza campo chimico	0.153147 s	0.037097 s
Tempo MCS complessivo	973.629472 s	47.636812 s

Reticolo: 400x400, MCS: 200, Numero Cellule: 40.
Matrice A: 25600000000, Vettore b 160000.

Tabella 4.4: Confronto dei tempi di esecuzione per la simulazione 3

Modulo	**C**	**CUDA**
Inizializza cellule	0.018060 s	0.003832 s
Risoluzione sistema lineare	16990.406143 s	357.869496 s
Calcolo campi elettrici	2.574847 s	0.007985 s
Aggiorna campo chimico	163.600027 s	42.542069 s
Normalizza campo chimico	1.269708 s	0.078279 s
Tempo MCS complessivo	17228.135654 s	437.953545 s

Reticolo: 800x800, MCS: 400, Numero Cellule: 80.
Matrice A: 409600000000, Vettore b: 640000.

Come evidenziato dalle tabelle 4.2, 4.3 e 4.4 le funzioni implementate in CUDA mostrano miglioramenti significativi nei tempi di esecuzione rispetto alle versioni in C, soprattutto con l'aumentare delle dimensioni del problema. È pur vero che, per le dimensioni del reticolo standard 200x200, la funzione per l'inizializzazione delle cellule

in C risulta leggermente più veloce del codice parallelo. Questa piccola discrepanza si riduce all'aumentare della scala del problema, dove la versione parallela diventa chiaramente più efficiente. La funzione di inizializzazione delle cellule viene eseguita una sola volta prima del ciclo di Monte Carlo, mentre le altre funzioni (risoluzione del sistema lineare, calcolo dei campi elettrici, aggiornamento e normalizzazione del campo chimico) fanno parte dell'algoritmo di Metropolis del metodo di Monte Carlo. Nel tempo complessivo del ciclo MCS sono incluse tutte le operazioni fondamentali del ciclo di Metropolis: associazione delle coppie cellula-vicino, calcolo dell'energia totale e decisione sull'accettazione della transizione di stato. Sono invece escluse le operazioni di salvataggio dello stato del reticolo e del campo chimico, le inizializzazioni e allocazioni delle strutture dati e qualsiasi altra routine non strettamente legata al Modello Cellulare di Potts.

4.2 Analisi dello speedup e della scalabilità

L'efficienza del programma parallelo implementato con CUDA è stata analizzata confrontando i tempi di esecuzione complessivi di un singolo Monte Carlo Step (MCS) rispetto alla versione sequenziale implementata in C. Lo speedup è stato calcolato utilizzando la seguente formula:

$$S = \frac{T_C}{T_{CUDA}}$$

dove T_C rappresenta il tempo di esecuzione sequenziale e T_{CUDA} il tempo di esecuzione parallelo.

Tabella 4.5: Speedup calcolato per ciascuna simulazione

Simulazione	**Speedup**
1	9,145
2	20,439
3	39,338

Per la Simulazione 1, con dimensioni ridotte del reticolo, il programma CUDA ha mostrato uno speedup pari a 9,145, indicando un'accelerazione significativa rispetto alla versione sequenziale. Tale risultato dimostra l'efficacia del parallelismo, nonostante l'overhead introdotto dal trasferimento dei dati e dalla sincronizzazione dei thread.

Nella Simulazione 2, con un reticolo di dimensioni maggiori, lo speedup è aumentato a 20,439, evidenziando un ulteriore miglioramento. Questo risultato riflette chiaramente come l'algoritmo parallelo sia in grado di scalare efficacemente con l'aumentare della complessità computazionale. Infatti, l'overhead diventa meno rilevante rispetto ai benefici apportati dal parallelismo, consentendo di sfruttare appieno la capacità di calcolo della GPU.

Per la Simulazione 3, con un reticolo ancora più grande, lo speedup raggiunge 39,338, dimostrando la capacità dell'implementazione CUDA di affrontare problemi su larga scala. Anche in questo caso, i benefici del parallelismo superano ampiamente i costi legati alla gestione delle risorse.

Questi risultati dimostrano che il programma è altamente scalabile, in particolare per problemi di grandi dimensioni, dove la quantità di lavoro parallelo supera di gran lunga i costi di overhead. L'incremento dello speedup con l'aumentare delle dimensioni del problema sottolinea la capacità dell'implementazione CUDA di gestire simulazioni su larga scala in modo molto più efficiente rispetto alla versione sequenziale.

Approfondimenti sulla scalabilità

La differenza di speedup tra le due simulazioni è un chiaro indicatore della scalabilità forte (*strong scaling*) del programma. Mentre la dimensione del reticolo cresce, il tempo di esecuzione parallelo aumenta in modo sub-lineare rispetto alla versione sequenziale, rendendo il programma particolarmente adatto per problemi di grandi dimensioni, come quelli richiesti per simulazioni realistiche di angiogenesi.

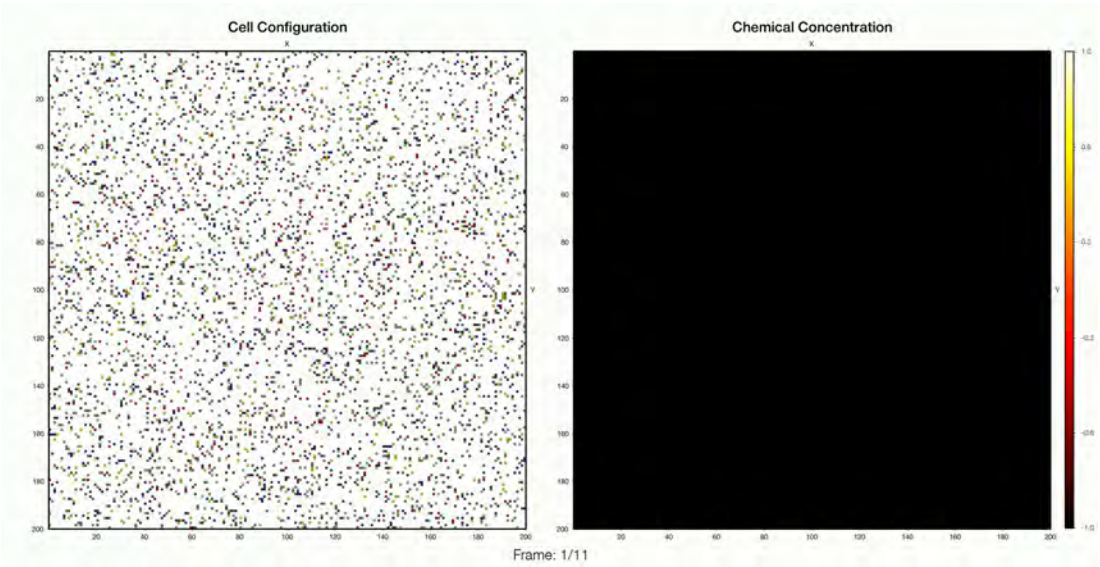

Figura 4.1: Frame 1 della simulazione CUDA

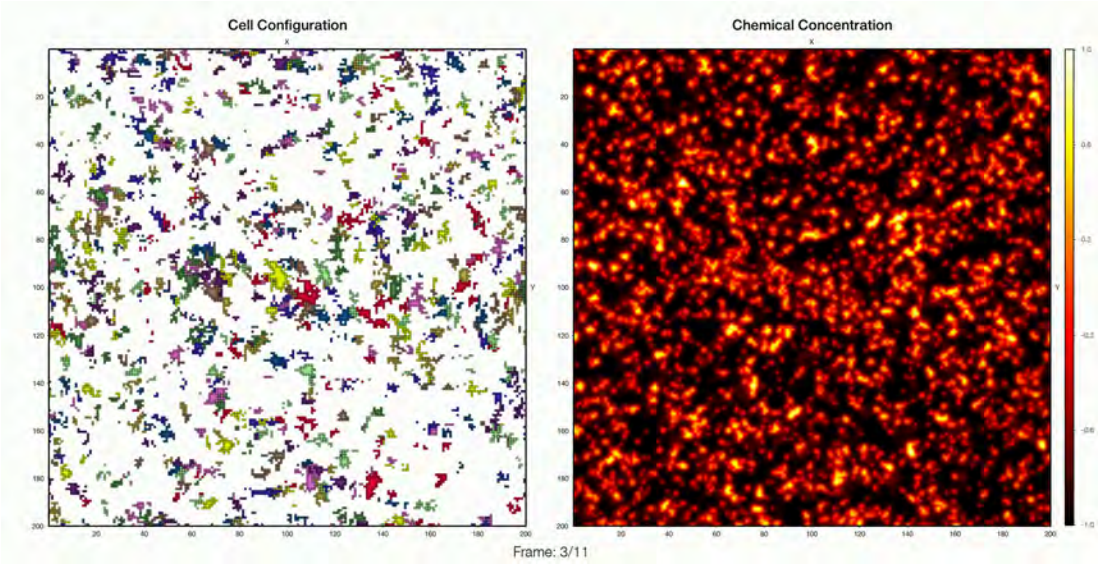

Figura 4.2: Frame 3 della simulazione CUDA

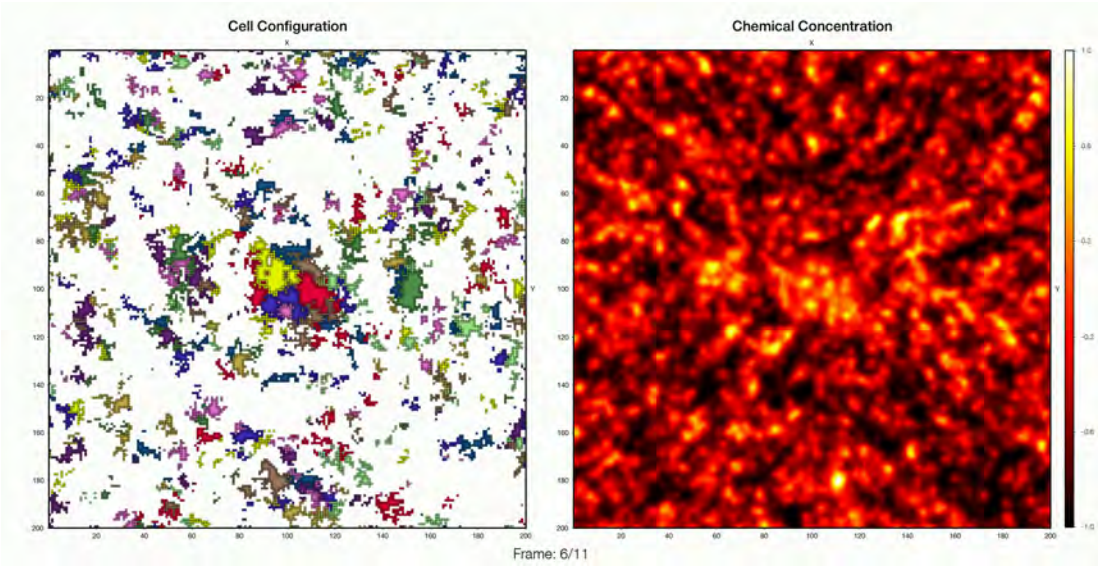

Figura 4.3: Frame 6 della simulazione CUDA

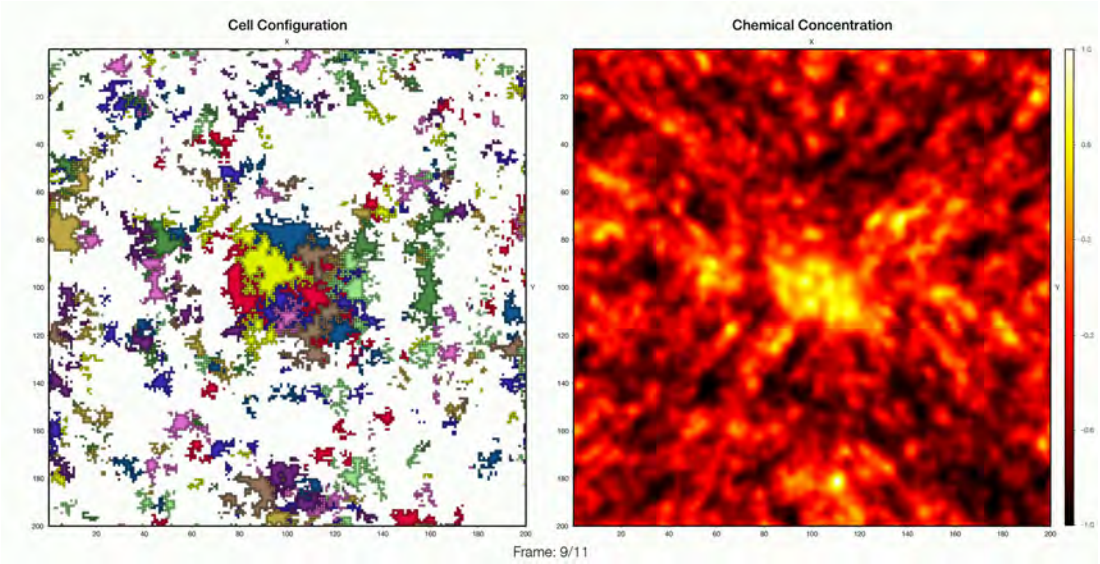

Figura 4.4: Frame 9 della simulazione CUDA

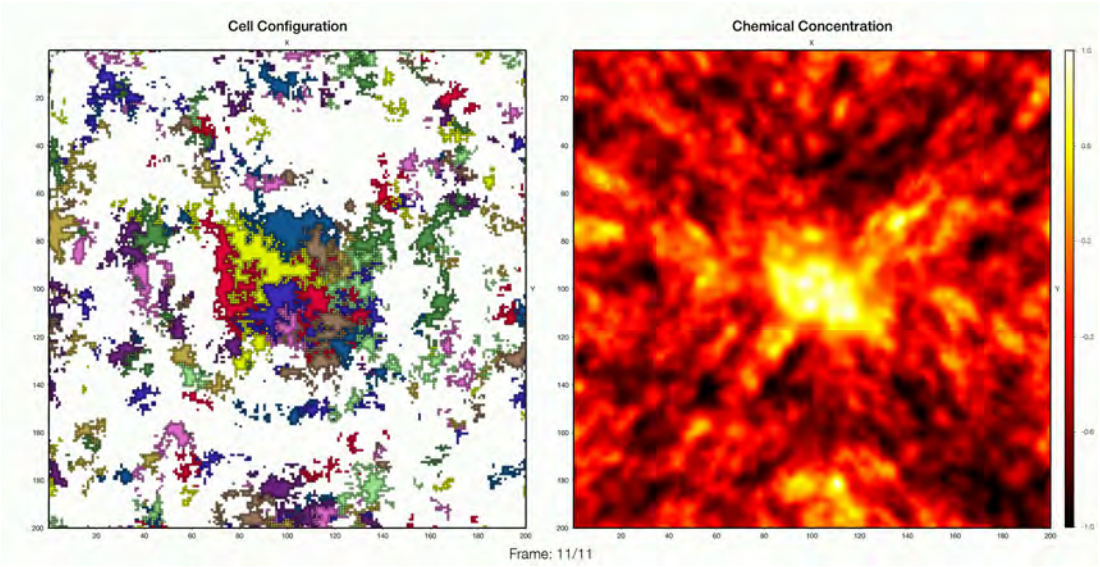

Figura 4.5: Frame 11 della simulazione CUDA

Accuratezza dei risultati

Nella prima fase del progetto, la parallelizzazione è stata inizialmente concentrata sulla parte computazionalmente più onerosa: la risoluzione del sistema lineare. Per garantire l'affidabilità dei risultati, durante lo sviluppo è stata adottata una strategia che prevedeva la risoluzione dello stesso sistema lineare all'interno del ciclo di Monte Carlo, sia in modalità sequenziale che parallela. I risultati delle due versioni sono stati confrontati a ogni iterazione per validare la correttezza dell'implementazione parallela. Questa strategia di verifica progressiva ha consentito di rilevare tempestivamente eventuali discrepanze tra i due approcci.

Naturalmente al completamento dello sviluppo, la doppia risoluzione è stata rimossa, consentendo di ottimizzare le prestazioni. Inoltre, la parallelizzazione è stata implementata in modo incrementale, adattando e testando gradualmente le funzioni del

codice originale. Questo approccio ha garantito che ogni fase della conversione fosse verificata accuratamente prima di procedere con ulteriori modifiche.

Efficienza computazionale

L'implementazione parallela ha dimostrato un miglioramento significativo in termini di efficienza computazionale rispetto alla versione sequenziale. Per la risoluzione del sistema lineare, è stata utilizzata la fattorizzazione LU sia nella versione sequenziale (SuiteSparse) che in quella parallela, implementata utilizzando cuDSS per la risoluzione del sistema lineare e CUB per la normalizzazione del campo chimico. Grazie a questa scelta, l'efficienza computazionale del codice parallelo ha superato di gran lunga quella del codice sequenziale, rendendo la GPU uno strumento essenziale per gestire simulazioni di grandi dimensioni.

I risultati mostrano che il codice parallelo raggiunge uno speedup significativo rispetto alla versione sequenziale:

- Per la Simulazione 1, lo speedup è pari a 9,145.
- Per la Simulazione 2, lo speedup raggiunge 20,439.
- Per la Simulazione 3, lo speedup aumenta ulteriormente fino a 39,338.

Questi valori evidenziano una chiara tendenza alla scalabilità forte (*strong scaling*), poiché l'incremento delle dimensioni del problema si traduce in una crescita ancora più marcata dello speedup. Questo comportamento riflette l'efficienza della parallelizzazione, in particolare per le simulazioni di grandi dimensioni, dove il lavoro computazionale supera ampiamente i costi di overhead.

Fattori di efficienza

Le prestazioni della versione parallela sono state ottimizzate riducendo l'overhead e sfruttando al meglio le risorse hardware disponibili:

- **Minimizzazione dell'overhead**: L'utilizzo di cuDSS e CUB ha consentito di ridurre al minimo le operazioni di copia tra CPU e GPU, migliorando così il tempo complessivo di esecuzione.
- **Ottimizzazione del parallelismo**: La suddivisione del lavoro tra i thread GPU è stata progettata per massimizzare il throughput, sfruttando tecniche di parallelizzazione avanzate.
- **Eliminazione progressiva delle dipendenze sequenziali**: Durante lo sviluppo, le sezioni critiche del codice sequenziale sono state gradualmente convertite in codice parallelo, aumentando l'efficienza complessiva.

4.2.1 Possibili miglioramenti futuri

L'attuale implementazione della simulazione rappresenta un punto di partenza solido per ulteriori sviluppi, sia dal punto di vista applicativo che computazionale. Di seguito, vengono delineate alcune possibili direzioni future per migliorare ed estendere il lavoro svolto:

Applicazione all'angiogenesi fisiologica e patologica

Una delle potenziali applicazioni più promettenti di questa simulazione è la possibilità di addestramento di un'intelligenza artificiale per analizzare l'evoluzione del processo simulato. In particolare, un modello di apprendimento automatico potrebbe essere utilizzato per identificare, in tempo reale, se lo sviluppo del sistema rappresenta un caso di angiogenesi fisiologica o patologica. Questa capacità predittiva potrebbe essere applicata a scenari reali, ad esempio per rilevare precocemente la formazione di un cancro e avviare tempestivamente un trattamento. Un tale sistema integrato di simulazione e analisi predittiva potrebbe rappresentare un importante strumento di supporto per la ricerca medica e oncologica.

Ottimizzazione della risoluzione del sistema lineare

Un'altra area di miglioramento riguarda la risoluzione del sistema lineare, attualmente basata sulla fattorizzazione LU con l'uso di cuDSS. Sebbene sia già estremamente efficiente, si potrebbe analizzare un approccio alternativo che potrebbe incrementare l'efficienza: l'utilizzo del metodo del Gradiente Coniugato (*Conjugate Gradient Method*). Questo metodo, particolarmente adatto a matrici sparse e al parallelismo offerto dalle GPU, presenta una complessità computazionale pari a $O(nnz)$, dove nnz è il numero di elementi non nulli della matrice. Inoltre, il gradiente coniugato si integra naturalmente con il contesto delle simulazioni, riducendo significativamente il costo computazionale rispetto ai metodi attuali.

Rimozione delle dipendenze sequenziali

Nonostante l'implementazione parallela, alcune parti del codice rimangono ancora sequenziali, limitando il potenziale massimo dello speedup ottenibile. Studi futuri potrebbero concentrarsi sull'eliminazione o sulla riduzione di queste dipendenze, esplorando strategie avanzate per rendere più parallele anche le operazioni attualmente sequenziali. Questo obiettivo potrebbe essere raggiunto con un'analisi approfondita delle dipendenze tra i dati e l'adozione di algoritmi che minimizzino le interazioni tra le varie parti del codice.

Estensione a simulazioni tridimensionali

Un'ulteriore direzione di sviluppo potrebbe consistere nell'estendere la simulazione a un contesto tridimensionale. Questo miglioramento consentirebbe di modellare fenomeni biologici con maggiore fedeltà, poiché molti processi angiogenici e tumorali avvengono in ambienti tridimensionali. Tuttavia, una tale estensione comporta un aumento significativo della complessità computazionale e richiederebbe ulteriori ottimizzazioni per garantire la scalabilità della simulazione.

Integrazione di modelli biologici avanzati

Infine, la simulazione potrebbe essere ampliata per includere ulteriori modelli biologici, come la chemotassi multipla, la meccanica delle cellule o l'interazione con il microambiente tumorale. Questi miglioramenti richiederebbero non solo un aumento della complessità algoritmica, ma anche una maggiore capacità di calcolo, rendendo

ancora più rilevante l'ottimizzazione dell'implementazione parallela.

Questi miglioramenti rappresentano le principali direzioni per il futuro sviluppo del progetto, con l'obiettivo di incrementare sia l'accuratezza biologica sia l'efficienza computazionale, e di espandere le potenziali applicazioni scientifiche e mediche della simulazione.

Bibliografia

[1] Gene M. Amdahl. “Validity of the single processor approach to achieving large scale computing capabilities”. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (1967), pp. 483–485.

[2] Alexander R. A. Anderson e Vito Quaranta. “Integrative mathematical oncology”. In: *Nature Reviews Cancer* 8.3 (2008), pp. 227–234.

[3] Gabriele Bergers e Douglas Hanahan. “Tumorigenesis and the angiogenic switch”. In: *Nature Reviews Cancer* 3.6 (2003), pp. 401–410.

[4] Kurt Binder e Dieter W. Heermann. *Monte Carlo Simulation in Statistical Physics: An Introduction*. 5th. Berlin, Germany: Springer, 2010. ISBN: 978-3-642-03162-5.

[5] Peter Carmeliet. “Angiogenesis in life, disease and medicine”. In: *Nature* 438.7070 (2005), pp. 932–936.

[6] Mark A. J. Chaplain e Alexander R. A. Anderson. “Mathematical modeling of tumor-induced angiogenesis: Network growth and structure”. In: *Cancer Treatment and Research* 82 (1996), pp. 177–196.

[7] Nan Chen et al. “A parallel implementation of the Cellular Potts Model for simulation of cell-based morphogenesis”. In: *Computer Physics Communications* 176.11-12 (2007), pp. 670–681.

[8] NVIDIA Corporation. *CUB Documentation*. 2023. URL: `https://nvlabs.github.io/cub/`.

[9] NVIDIA Corporation. *CUDA Best Practices Guide*. 2023. URL: `https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/`.

[10] NVIDIA Corporation. *cuDSS Library Documentation*. 2023. URL: `https://docs.nvidia.com/cuda/cuDSS/`.

[11] NVIDIA Corporation. *cuDSS Preview Documentation*. 2023. URL: `https://developer.nvidia.com/cudss-preview`.

[12] NVIDIA Corporation. *Grace Hopper Superchip Overview*. 2023. URL: `https://www.nvidia.com/en-us/data-center/grace-hopper/`.

[13] Pasquale De Luca e Livia Marcellino. “Incorporating Chemotaxis into the Cellular Potts Model: Mathematical Foundations”. In: *Extension CPM* (set. 2024). Preprint.

[14] James W Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

[15] Jack Dongarra. “HPC and the evolution of supercomputers”. In: *Communications of the ACM* 62.11 (2019), pp. 67–75.

[16] Napoleone Ferrara e Robert S Kerbel. “Angiogenesis as a therapeutic target”. In: *Nature* 438.7070 (2005), pp. 967–974.

[17] Michael J Flynn. “Some computer organizations and their effectiveness”. In: *IEEE Transactions on Computers* C-21.9 (1972), pp. 948–960.

[18] James A. Glazier e François Graner. “Simulation of the differential adhesion driven rearrangement of biological cells”. In: *Physical Review E* 47.3 (1993), pp. 2128–2154.

[19] Gene H Golub e Charles F Van Loan. *Matrix Computations*. Johns Hopkins University Press, 2013.

[20] François Graner e James A. Glazier. “Simulation of biological cell sorting using a two-dimensional extended Potts model”. In: *Physical Review Letters* 69.13 (1992), pp. 2013–2016.

[21] Douglas Hanahan e Robert A Weinberg. “Hallmarks of cancer: the next generation”. In: *Cell* 144.5 (2011), pp. 646–674. DOI: https://10.1016/j.cell.2011.02.013.

[22] John L Hennessy e David A Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2011.

[23] John L Hennessy e David A Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2017.

[24] Nicholas J Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002.

[25] Rakesh K Jain. “Normalization of tumor vasculature: an emerging concept in antiangiogenic therapy”. In: *Science* 307.5706 (2005), pp. 58–62.

[26] M. H. Kalos e P. A. Whitlock. *Monte Carlo Methods*. 2nd. Weinheim, Germany: Wiley-VCH, 2008.

[27] Evelyn F. Keller e Lee A. Segel. “Model for Chemotaxis”. In: *Journal of Theoretical Biology* 30.2 (1971), pp. 225–234.

[28] David B Kirk e Wen-Mei W Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2017.

[29] S. Kirkpatrick, C. D. Gelatt e M. P. Vecchi. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680. DOI: 10.1126/science.220.4598.671.

[30] Steven R. McDougall, Alexander R. A. Anderson e Mark A. J. Chaplain. “Mathematical modeling of dynamic adaptive tumour-induced angiogenesis: Clinical implications and therapeutic targeting strategies”. In: *Journal of Theoretical Biology* 241.3 (2006), pp. 564–589.

[31] Roeland M. H. Merks et al. “Cell elongation is key to in silico replication of in vitro vasculogenesis and subsequent remodeling”. In: *Developmental Biology* 289.1 (2006), pp. 44–54.

[32] Nicholas Metropolis et al. “Equation of State Calculations by Fast Computing Machines”. In: *Journal of Chemical Physics* 21.6 (1953), pp. 1087–1092. DOI: 10.1063/1.1699114.

[33] Netlib LAPACK Project. *Computational Complexity of LAPACK Algorithms*. 2024. URL: https://www.netlib.org/lapack/complexity.html.

[34] Netlib LAPACK Project. *LAPACKE Interface Documentation*. 2024. URL: https://www.netlib.org/lapack/lapacke.html.

[35] NVIDIA. *NVIDIA GPU Architecture Overview*. Disponibile su: https://developer.nvidia.com/gpu-architecture. 2020.

[36] NVIDIA Corporation. *CUDA cuSOLVER Documentation*. 2024. URL: https://docs.nvidia.com/cuda/cusolver/index.html.

[37] NVIDIA Corporation. *CUDA cuSPARSE Documentation*. 2024. URL: https://docs.nvidia.com/cuda/cusparse/index.html.

[38] NVIDIA Corporation. *CUDA Programming Guide*. 2024. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[39] NVIDIA Corporation. *CUDA Toolkit Documentation*. 2024. URL: https://docs.nvidia.com/cuda/.

[40] OpenCV Team. *cv::Mat - The Basic Image Container*. 2024. URL: https://docs.opencv.org/4.x/d3/d63/classcv_1_1Mat.html.

[41] OpenCV Team. *OpenCV Documentation*. 2024. URL: https://docs.opencv.org/.

[42] TOP500 Organization. *TOP500 List - November 2024*. Accessed December 12, 2024. 2024. URL: https://www.top500.org.

[43] Shayn M. Peirce. "Computational and mathematical modeling of angiogenesis". In: *Microcirculation* 15.8 (2008), pp. 739–751.

[44] Katarzyna A. Rejniak. "A Single-Cell Approach in Modeling the Dynamics of Tumor Microregions". In: *Mathematical Biosciences and Engineering* 4.3 (2007), pp. 171–187.

[45] Domenico Ribatti. "History of research on angiogenesis". In: *Chemical Immunology and Allergy* 89 (2007), pp. 1–14.

[46] Domenico Ribatti. "Tumor angiogenesis: a historical review". In: *International Journal of Oncology* 37.1 (2010), pp. 1–12.

[47] NVIDIA Documentation Team. "CUDA Implementation of QR Factorization Using Householder Transformations". In: *NVIDIA Developer Blog* (2020).

[48] Timothy A. Davis. *SuiteSparse Documentation*. 2024. URL: http://faculty.cse.tamu.edu/davis/suitesparse.html.

[49] Andrea Tosin, Davide Ambrosi e Luigi Preziosi. "Mechanics and chemotaxis in the morphogenesis of vascular networks". In: *Bulletin of Mathematical Biology* 68 (2006), pp. 1819–1836.

[50] Lloyd N Trefethen e David Bau III. *Numerical Linear Algebra*. SIAM, 1997.

[51] O.C. Zienkiewicz e R.L. Taylor. *The Finite Element Method: Its Basis and Fundamentals*. 5th. Butterworth-Heinemann, 2000.