ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria e Architettura

Corso di Laurea in Ingegneria e Scienze Informatiche

Simulazione del Sistema Immunitario Umano tramite il Framework FLAME GPU

Relatore:
Chiar.mo Prof.
Moreno Marzolla

Presentata da:
Samuele Casadei

IV Sessione di laurea - Febbraio 2026
A.A. 2025-2026

Sommario

In questa tesi viene descritta l'implementazione di un modello computazionale volto alla simulazione del sistema immunitario umano, basato su una variante del modello *Celada-Seiden*. Il modello è stato sviluppato usando FLAME GPU, un framework per simulazioni parallele ad agenti su GPU basato su CUDA, e verrà confrontato con una versione in CUDA puro. Lo scopo della tesi è di valutare prestazioni e complessità delle due versioni, e capire se FLAME GPU sia una valida alternativa a CUDA nelle simulazioni ad agenti. La tesi introdurrà il modello utilizzato come base e il framework di FLAME GPU. In seguito si tratteranno la progettazione e l'implementazione; infine verrà verificata la correttezza dei programmi e saranno messi a confronto i tempi di esecuzione.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Contesto Scientifico

Il sistema immunitario è una rete integrata composta da cellule e molecole che interagiscono dinamicamente tra loro per proteggere l'organismo dalle infezioni. È in grado di rilevare l'introduzione di agenti potenzialmente pericolosi nel corpo e provoca la risposta immunitaria necessaria alla loro eliminazione, evolvendosi dopo ogni risposta.
Lo sviluppo di modelli di simulazione ad agenti del sistema immunitario rappresenta un approccio consolidato per lo studio dell'immunologia. Tali modelli permettono di esplorare scenari difficili da riprodurre sperimentalmente e forniscono un supporto significativo alla ricerca su vaccini e malattie autoimmuni.

1.2 Obiettivi

L'obiettivo di questa tesi è verificare la praticabilità dell'utilizzo di un framework di alto livello, come FLAME GPU, per le simulazioni di modelli immunologici ad agenti, confrontandone le prestazioni con un'implementazione equivalente sviluppata in CUDA di basso livello.
Da questo obiettivo principale si definiscono più sotto-obiettivi:

- Implementare il modello ad agenti in FLAME GPU.
- Validare la correttezza del modello.

- Confrontare prestazioni e complessità rispetto al codice CUDA esistente.

1.3 Modellazione ad Agenti

L'ABM (Agent-Based Modelling) è una classe di modelli computazionali volta a simulare il comportamento di unità autonome, dette agenti. Ciascun agente interagisce con altri agenti e l'ambiente circostante, operando secondo un insieme di regole predefinito.

1.3.1 Agenti

Gli agenti sono entità autonome e capaci di prendere decisioni, che simulano il comportamento dei singoli componenti all'interno di un sistema. Possono essere reattivi, se rispondono ai cambiamenti dell'ambiente, o proattivi, se prendono decisioni basate su stati interni o obiettivi. Alcuni modelli includono agenti capaci di adattare il proprio comportamento nel tempo, utilizzando algoritmi genetici e strategie evolutive.

1.3.2 Componenti della Modellazione ad Agenti

- **Agenti**: entità indipendenti con attributi e regole che ne definiscono il comportamento.
- **Ambiente**: lo spazio in cui gli agenti esistono e interagiscono.
- **Regole di Interazione**: linee guida che definiscono le interazioni possibili degli agenti tra loro e con l'ambiente.
- **Comportamenti Emergenti**: fenomeni o pattern globali che nascono spontaneamente dalle interazioni locali tra gli agenti, senza essere programmati esplicitamente.

1.3.3 Vantaggi della Modellazione ad Agenti

La modellazione ad agenti è molto utile per rappresentare sistemi dove il comportamento delle singole entità non segue un percorso lineare. Le interazioni tra agenti portano spesso a comportamenti non lineari, cicli di feedback (processo nel quale un sistema usa il suo stesso output per influenzare comportamenti futuri) ed evoluzioni adattive.

Nella progettazione software, la modellazione ad agenti supporta in modo naturale un approccio "*bottom-up*": si definisce la logica dei singoli agenti e, attraverso le loro interazioni, si ottiene il comportamento complessivo del sistema in modo spontaneo.
La modellazione ad agenti permette di testare molti scenari di casi limite in ambiente virtuale prima di applicarli in ambito sperimentale, salvando tempo, denaro e riducendo i rischi.

1.3.4 Criticità della Modellazione ad Agenti

Le più importanti limitazioni per l'ABM comprendono:

- **Raccolta di Dati**: per far si che una simulazione dia esiti più verosimili possibili è necessario raccogliere dati estremamente accurati sul comportamento dei singoli agenti.
- **Risorse Computazionali**: Quando il numero di agenti ammonta a migliaia e milioni, simulare il comportamento di ogni singolo agente e le interazioni con le altre entità richiede dell'hardware potente e tempo per eseguire le simulazioni.

Capitolo 2

Modello Computazionale

2.1 Introduzione al Modello

Il modello Celada-Seiden è una descrizione logica dei meccanismi che regolano la risposta immunitaria adattativa, sia umorale (anticorpi) sia cellulare, a un antigene genericamente definito. Si tratta di un insieme di regole e processi biologici formalizzati in modo da poter essere simulati su un computer. Il modello è stato sviluppato dall'immunologo Franco Celada e dal fisico Philip E. Seiden [1]. L'approccio scelto per simulazione è quello di un *automa cellulare*, un sistema dinamico la cui evoluzione è descritta da interazioni locali degli agenti al suo interno, ed è discreto sia nello spazio che nel tempo. Le regole che definiscono un automa cellulare sono le seguenti:

1. È costituito da una griglia di celle.

2. Si evolve a timestep discreti.

3. Ogni cella può assumere un numero finito di valori.

4. Il valore di ogni cella si evolve secondo delle regole deterministiche.

5. Le regole dell'evoluzione di ogni cella si basa sui valori assunti dalle celle adiacenti.

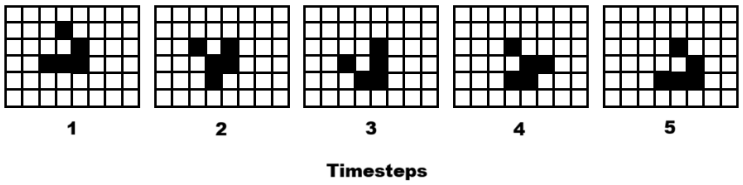

Figura 2.1: Esempio di automa cellulare nel corso dei timestep, Game of Life di John Conway [1, Fig.1]

Il modello Celada-Seiden introduce i seguenti cambiamenti alle regole:

- Il valore di ogni cella si evolve secondo delle regole **probabilistiche**.
- Le regole che determinano l'evoluzione di ogni cella si basano sul valore assunto dalla cella stessa.
- **Nuova regola**: al termine di ogni timestep le entità possono spostarsi nelle celle adiacenti.

2.2 Entità e Interazioni Definite

Il modello Celada-Seiden non definisce il funzionamento dell'intero sistema immunitario, ma emula la parte più cruciale della risposta umorale, pertanto il numero di entità modellate è limitato.

2.2.1 Entità

Le entità descritte nel modello Celada-Seiden sono le seguenti:

- **Antigene**: antigene generalizzato: una molecola in grado di generare una risposta adattiva del sistema immunitario, la sua presenza nel corpo è anormale, ma non ha nessun effetto malevolo specifico.
- **Anticorpo**: una molecola proteica (*immunoglobulina*) prodotta nel corso di una reazione immunitaria. Si lega ad un antigene specifico, disattivandolo e formando un Complesso Immune.

- **Complesso Immune**: una molecola formata dal legame di più antigeni ad anticorpi.

- **Cellula-A**: cellula appartenente alla categoria dei macrofagi, si occupa di fagocitosi e distruzione di agenti estranei presenti nel corpo. Si lega agli antigeni con bassa affinità tramite i recettori presenti sulla membrana (MHC), si lega ai complessi immuni con alta affinità; dopo aver assimilato un antigene ne presenta i peptidi sulla superficie.

- **Linfocita-B**: si lega agli antigeni in base alla compatibilità tra peptidi e recettori, in seguito presenta i peptidi degli antigeni sulla superficie, può essere stimolato dai linfociti-T per produrre cellule memoria e anticorpi della stessa specificità dell'antigene assimilato.

- **Linfocita-T**: riconosce i peptidi sulle cellule-A e i linfociti-B, stimola gli ultimi a produrre anticorpi e memoria.

2.2.2 Interazioni

Le interazioni avvengono tra entità che occupano posizioni adiacenti all'interno della griglia, quelle definite nel modello sono le seguenti:

- **Linfociti B - Antigeni**: I linfociti tentano di legarsi con gli antigeni che presentano un'adeguata compatibilità dei recettori, processando l'antigene e presentandone i peptidi.

- **Anticorpi - Antigeni**: Gli anticorpi tentano di legarsi agli antigeni tramite i recettori; legandosi all'antigene lo disattivano e formano un complesso immune.

- **Cellule A - Complessi immuni**: Le cellule si legano ai complessi immuni fagocitandoli e processandoli, presentandone i peptidi in superficie.

- **Cellule A - Antigeni**: Le cellule fagocitano gli antigeni e li processano, presentandone i peptoidi in superficie.

- **Linfociti T - Linfociti B**: I linfociti T, riconoscendo peptidi di antigene presentati da linfociti B o Cellule A, invia un segnale chimico (rilascio di chitochine) ai linfociti B per stimolarli alla produzione di anticorpi e cellule memoria.

2.3 Tipologia di Simulazione

Come detto in precedenza, l'obiettivo della tesi è quello di realizzare una versione FLAME GPU che sia equivalente alla versione CUDA precedentemente implementata. Il modello adottato dalla versione CUDA è una versione semplificata del modello Celada-Seiden descritto, in quanto:

- Le Cellule-A sono escluse dall'implementazione.
- Gli Antigeni, una volta legati agli Anticorpi, non divengono Complessi immuni; in quanto possono essere rimossi solo da Cellule-A.
- Gli Anticorpi non presentano un recettore, legano con gli Antigeni indiscriminatamente.
- I Linfociti-T attivano i Linfociti-B senza effettuare il riconoscimento dei peptidi, specularmente, i Linfociti-B non mostrano i peptidi quando processano un antigene; questa interazione viene implementata come una transizione di stato del Linfocita-B.

Capitolo 3

Tecnologie utilizzate

3.1 GPU e Calcolo Parallelo

La programmazione parallela consiste nell'esecuzione contemporanea di una o più istruzioni di codice su molteplici unità di calcolo con lo scopo di aumentare le prestazioni di calcolo dell'elaboratore.

Un programma può essere definito "parallelo" quando al suo interno esiste anche una sola porzione di codice eseguita simultaneamente su più unità di calcolo. Durante l'esecuzione di una sezione parallela di codice, l'elaboratore suddivide il processo in multiple unità di lavoro, ciascuna con il compito di eseguire autonomamente la propria porzione di codice.

L'esigenza di architetture parallele è nata dal raggiungimento dei limiti della "legge di Moore": nel 1965 Gordon Moore ipotizzò che il numero di transistor nei microprocessori sarebbe raddoppiato ogni 12 mesi circa [9]; inserire un maggior numero di transistor all'interno dei microprocessori comporta un maggior consumo elettrico, che può originare surriscaldamento e interferenza elettromagnetica, portando a instabilità operativa e a fenomeni di degrado termico. La soluzione a questo limite è stata trovata nello sviluppo di sistemi paralleli, che sfruttano i core fisici dei microprocessori e, quando disponibile, le unità di elaborazione virtuali per eseguire più istruzioni in parallelo, aumentando le prestazioni senza incrementare il numero di transistor.

Esistono diverse tecniche per gestire il flusso di dati e istruzioni, queste tecniche sono descritte dalla Tassonomia di Flynn:

- **SISD (Single Instruction stream Single Data stream)**: l'architettura basilare descritta dal modello di Von Neumann, le istruzioni sono eseguite sequenzialmente su un singolo flusso di dati.
- **SIMD (Single Instruction stream Multiple Data stream)**: livello più comune di parallelismo, dove più unità di calcolo eseguono la stessa operazione su flussi di dati diversi.
- **MISD (Multiple Instruction stream Single Data stream)**: istruzioni diverse su un singolo flusso di dati, non utilizzato in pratica.
- **MIMD (Multiple Instruction stream Multiple Data stream)**: queste architetture permettono l'esecuzione di multiple istruzioni operanti nel loro flusso di dati personale.

Per quanto riguarda le tecniche MIMD, si possono suddividere in tre sottocategorie di modelli:

- Modelli a memoria condivisa: tutte le unità di esecuzione operano tutte sulla stessa memoria del programma; un esempio è OpenMP.
- Modelli a memoria distribuita: ogni unità ha la propria area di memoria su cui opera individualmente; un esempio è MPI.
- Graphics Processing Unit (GPU): sfrutta l'utilizzo di un microprocessore con un elevatissimo numero di unità di esecuzione, ogni unità di lavoro ha una propria memoria e ha accesso a una memoria condivisa fornita dal microprocessore; un esempio è CUDA.

3.2 CUDA

Compute Unified Device Architecture (CUDA) è una piattaforma di programmazione utilizzata per il calcolo parallelo sfruttando l'architettura hardware delle GPU NVIDIA. Le GPU sono dispositivi di elaborazione specializzati nel *rendering*, ovvero la generazione di immagini digitali (sia 2D che 3D); questi processi, generalmente, operano eseguendo la medesima istruzione su enormi quantità di dati (pixel e matrici), dunque richiedono un elevato numero di unità di esecuzione per scalare adeguatamente.

All'interno della GPU sono presenti un grande numero di Streaming Multiprocessor (SM), i quali a loro volta contengono un certo numero di Streaming Processor (SP); questi rappresentano i "core" della GPU, sono molto meno

complesse dei core delle CPU, occupano meno area in silicio e consumano meno energia; questo permette di inserirne migliaia all'interno di un singolo microprocessore. Le singole unità lavorative sono dette *thread* e sono allocate in griglie tridimensionali, formate da un certo numero di blocchi; i blocchi sono assegnati agli SM, che ne gestiscono l'esecuzione.

Durante l'esecuzione, i thread di un blocco vengono raggruppati in gruppi di 32 thread, detti anche *warp*. All'interno di un warp, tutti i thread eseguono la stessa operazione contemporaneamente.

I programmi CUDA suddividono le istruzioni del programma in due sottoprogrammi diversi, in base al microprocessore che le esegue:

- **Host**: programma destinato ad eseguire su CPU.
- **Device**: programma che esegue esclusivamente su GPU.

I due sottoprogrammi eseguono in contesti completamente diversi, su memorie diverse. Il programma host esegue per primo, alloca le risorse per il programma device e lanciarlo tramite delle apposite funzioni, dette anche *kernel*.

3.3 Introduzione a FLAME GPU

FLAME GPU è un acronimo che sta per *Flexible Large scale Agent Modelling Environment for the GPU*. Si tratta di un framework che permette di implementare modelli ad agenti sfruttando il parallelismo massivo offerto da CUDA [2].

Il framework consente di implementare modelli ad agenti ad alto livello, concentrandosi sulla definizione del comportamento degli agenti, senza dover gestire direttamente i dettagli legati alla programmazione GPU e alle strategie di ottimizzazione a basso livello. In particolare si fa riferimento alla versione 2.x di FLAME GPU, in cui il modello viene definito in codice C++, includendo concetti fondamentali della modellazione basata su agenti quali la presenza di più tipologie di agenti, la comunicazione tramite messaggi e la gestione dinamica della nascita e della morte degli agenti. Il framework si occupa automaticamente della traduzione di tali specifiche in codice CUDA ottimizzato, consentendo l'esecuzione efficiente delle simulazioni sulla GPU.

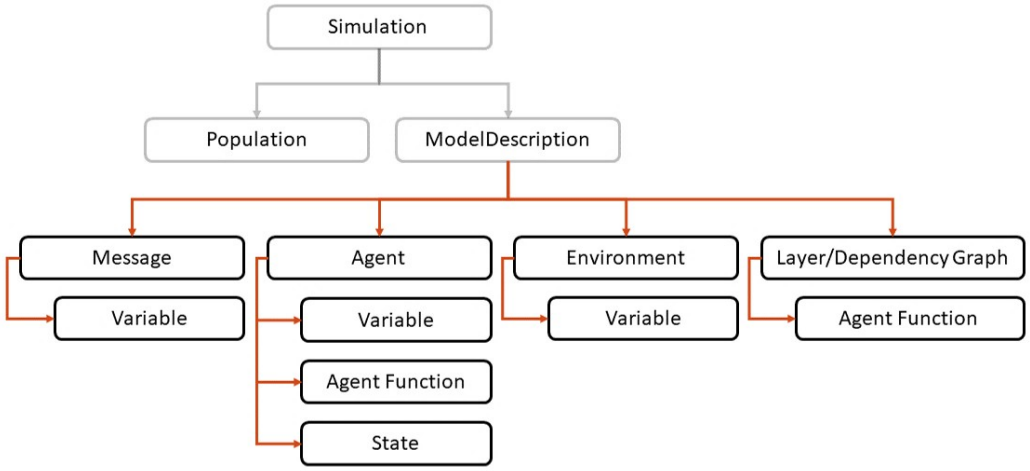

Figura 3.1: Gerarchia dei componenti di una simulazione FLAME GPU [8, Fig.4]

.

3.3.1 Componenti di un Modello FLAME GPU

I componenti principali di una simulazione ad agenti FLAME GPU sono i seguenti:

- **Agenti**: elemento centrale della simulazione, ad ogni agente possono assere assegnati: delle variabili interne, delle proprie *agent-function* che ne dettano il comportamento e molteplici stati tra cui variare nel corso della simulazione.
- **Ambiente** (o environment): l'environment rappresenta l'insieme di proprietà esterne che influiscono sul comportamento degli agenti; le variabili dell'ambiente (dette proprietà, appunto) sono esclusivamente *read-only* per gli agenti, tuttavia possono essere modificate nel corso della simulazione tramite funzioni host.

- **Messaggi**: la comunicazione tra agenti viene gestita tramite l'invio e la ricezione di messaggi; ciascun messaggio contiene delle variabili utilizzate per trasmettere informazioni tra gli agenti. Gli agenti possono inviare e ricevere messaggi solo all'interno delle proprie agent-function; la modalità con cui si accede a un tipo di messaggio è dettata dalla strategia di comunicazione adottata dal messaggio stesso.

- **Layer**: i layer definiscono l'ordine di esecuzione delle agent-function dei vari agenti, funzioni all'interno dello stesso layer eseguono contemporaneamente.

I messaggi in FLAME GPU possono seguire diverse strategie di comunicazione:

- Brute Force: ogni messaggio viene inviato in *broadcast* e reso accessibile ad ogni agente per la durata del timestep, crea una comunicazione *all-to-all* tra gli agenti ed è il più costoso in termini di risorse.

- Spatial: ogni agente accede solo i messaggi in un raggio specificato, può essere specificato sia in due che in tre dimensioni.

- Bucket: gli agenti accedono ai messaggi mappati su una specifica chiave intera.

- Array: i messaggi sono inseriti in un array, che può essere a una, due o tre dimensioni.

Il mezzo di diffusione dei messaggi sono le agent function; questo possono ricevere dati da un messaggio di input e inviare dati tramite un messaggio di output.
Per quanto riguarda la parallelizzazione offerta dai layer, ci sono dei vincoli imposti da FLAME GPU:

1. Lo stesso agente non può eseguire più di un agent function per layer.

2. Due agenti diversi non possono inviare messaggi dello stesso tipo nello stesso layer.

3. Ogni invio di una tipologia di messaggio sovrascrive completamente gli invii precedenti.

4. I messaggi prodotti in un layer diventano visibili solo nei layer successivi: non è consentito leggere nello stesso layer in cui vengono prodotti.

5. L'ordine di esecuzione delle agent function all'interno dello stesso layer non è garantito.

6. Variabili interne agli agenti modificate in un layer diventano visibili solo nei layer successivi.

7. Gli agenti creati o rimossi durante un layer diventano effettivi solo a partire dal layer successivo.

In questo framework sono comprese anche delle funzioni che vengono eseguite su CPU, ovvero le *host function*. Le host function possono essere di tre tipi specifici:

1. **Init function**: Un host function eseguita solo una volta, prima dell'inizio della simulazione; utilizzata per creare gli agenti iniziali e inizializzare lo stato globale della simulazione.

2. **Step function**: si tratta di una normale host function inserita nello step loop. Tali funzioni generalmente si occupano di scritture su file e visualizzazione; le step function sono molto costose in termini di tempo, richiedono una sincronizzazione tra CPU e GPU e non possono essere parallelizzate, per questo motivo dovrebbero essere eseguite sporadicamente nel corso della simulazione.

3. **Exit condition**: è una host function particolare, eseguita al termine di ogni timestep, esegue un controllo e può terminare con due possibili espressioni: `flamegpu::EXIT` e `flamegpu::CONTINUE`; "exit" comporta la terminazione del programma, mentre "continue" permette di proseguire.

3.3.2 Differenze tra FLAME GPU e CUDA

Si può sfruttare appieno l'efficienza di CUDA se:

1. Si ha una buona conoscenza dell'architettura hardware della GPU che si sta utilizzando.

2. Vengono scelte le strategie di ottimizzazione migliori per sfruttare al massimo la memoria condivisa tra i thread.

Un programmatore esperto in programmazione GPU riuscirebbe a sfruttare le piene potenzialità della piattaforma, con risultati che sarebbero difficilmente raggiungibili usando FLAME GPU. Oltre a questo, la programmazione

CUDA è impegnativa, in quanto richiede gestione manuale della memoria GPU, con ottimizzazione degli accessi alla memoria e gestione manuale delle race condition sui dati, copiare variabili da host a device e viceversa; Vanno definiti e lanciati i kernel ragionando in termini di thread, warp e blocchi. In sostanza, si tratta di una gestione esplicita del parallelismo a basso livello, che offre massima flessibilità (che può comportare anche la massima efficienza), ma anche un'elevata complessità concettuale.

Al contrario, FLAME GPU nasconde la complessità di CUDA dietro ad un modello concettuale di alto livello; con FLAME GPU, lo sviluppo si concentra attorno alla definizione degli agenti e alle loro regole di interazione, il parallelismo è implicito e avviene "dietro le quinte". Questa tecnologia è rilevante in ambito di ricerca perché l'implementazione dei modelli e eventuali modifiche delle regole richiedono poco tempo rispetto allo sviluppo CUDA, questo permette più sperimentazioni in minor tempo.

FLAME GPU converte la definizione di modelli in codice CUDA già ottimizzato da esperti di programmazione GPU, in cui si fa uso di coalescenza degli accessi alla memoria, gestione di race condition sui dati e raggruppamento dei dati in memoria per favorire accessi a celle contigue di memoria [3].

3.4 Scelta di FLAME GPU

È stato scelto FLAME GPU in quanto rappresenta un'opzione particolarmente valida nell'ambito della ricerca sperimentale e permette la realizzazione di modelli di simulazione ad agenti in parallelo senza richiedere competenze avanzate di programmazione GPU, pur garantendo i medesimi benefici in termini di tempi di esecuzione di un programma parallelo. Oltre a questo, i tempi di sviluppo di un modello ed eventuali aggiustamenti e modifiche delle dinamiche di interazione richiedono meno tempo rispetto ad un programma CUDA, in cui potrebbe essere necessario stravolgere l'implementazione ad ogni cambiamento; questo rende possibile sperimentare più fenomeni in meno tempo senza rinunciare all'efficienza e alle elevate prestazioni [8, Fig.5].

Capitolo 4

Progettazione e Implementazione

4.1 Struttura del modello

In questa sezione verranno descritte le dinamiche di interazioni tra gli agenti e come sono state tradotte le entità del modello nel progetto FLAME GPU.

4.1.1 Agenti

Ogni agente nel modello presenta le seguenti variabili:

- **x** e **y**: delle variabili positive a virgola mobile necessarie per determinare la posizione di un agente nel sistema.
- **vx** e **vy**: delle variabili a virgola mobile per monitorare la variazione di velocità lungo gli assi cartesiani.
- **mass**: un dato che rappresenta la massa dell'agente, necessaria per calcolare la nuova velocità.

Gli agenti necessitano di queste variabili per potersi spostare nel sistema seguendo il *moto browniano*, cioè il moto disordinato, osservabile al microscopio, di particelle sufficientemente piccole presenti nei fluidi o nelle sospensioni [7, pp. 14-15].

Gli agenti che partecipano attivamente alle interazioni presentano la variabile

has_interacted: un dato booleano che serve per tenere traccia dell'interazione di un agente nel corso del timestep.

Gli agenti implementati sono i seguenti:

- **Antigeni**: sono agenti che interagiscono in modo passivo per l'intera simulazione, presentano la variabile **epitope** che emula il comportamento dei peptidi dell'antigene.
- **Anticorpi**: sono agenti che interagiscono attivamente, legandosi agli antigeni ed eliminandoli.
- **Linfociti-B**: sono agenti che interagiscono attivamente durante la simulazione, cambiano lo stato nel corso delle iterazioni, pertanto contengono una variabile **state** per tenerne traccia; hanno una variabile **receptor** usata per legare con gli antigeni, ogni volta che legano con un antigene il recettore è sottoposto ad un'*ipermutazione* [7, p.14].
- **Linfociti-T**: sono agenti attivi e non presentano un recettore.

Per simulare il comportamento dei recettori vengono usati degli array di byte. L'affinità tra recettori viene calcolata utilizzando la formula *Bernaschi-Castiglione*; tramite questa procedura è possibile comparare due stringhe binarie ottenendo, come risultato, un valore che esprime la probabilità dei due recettori di legare [7, p.13].

4.1.2 Environment

L'environment rappresenta lo stato globale della simulazione, condiviso da tutti gli agenti e accessibile (prevalentemente in sola lettura) durante l'esecuzione; esso immagazzina valori tipizzati detti *property*.
Viene riportato di seguito l'elenco delle property utilizzate nella simulazione:

- *total_steps*: un intero positivo che contiene il numero di iterazioni che la simulazione deve svolgere prima di terminare; è necessario per tenere traccia della proporzione tra timestep corrente e timestep totali, in modo da programmare la stampa di grafici "relativamente" allo svolgimento della simulazione (ad esempio al 10%, 20% e 30%) piuttosto che utilizzare un numero fisso di timestep.
- *width* e *height*: parametri a virgola mobile che determinano la dimensione della griglia continua all'interno della quale possono esistere agenti; è utilizzata dalle funzioni di generazione e movimento degli agen-

ti per evitare che delle entità finiscano oltre i limiti dell'ambiente di simulazione.

- *B_cell_num*, *T_cell_num* e *antigen_num*: numero di linfociti B, linfociti T e antigeni, necessari per l'inizializzazione degli agenti nella simulazione.

- *interaction_radius*: raggio di interazione a virgola mobile utilizzato dalle agent function per determinare con quali entità è possibile interagire.

4.1.3 Messaggi e Agent Function

Il modello si basa esclusivamente sulla strategia di comunicazione **Spatial 2D**. I messaggi di questa tipologia richiedono obbligatoriamente le variabili x e y, che rappresentano la posizione dell'agente mittente; pertanto, nella descrizione del contenuto dei messaggi definiti, tali variabili verranno omesse in quanto implicite.

La dinamica di scambio dei messaggi all'interno del modello nel corso di un timestep comincia con l'invio della posizione da parte di ogni antigene insieme all'identificatore (utilizzato in messaggi successivi) e al recettore; in seguito, gli anticorpi rilevano la presenza di antigeni nelle vicinanze e, nel caso in cui ne trovassero, inviano un messaggio di eliminazione contenente l'identificatore dell'antigene da eliminare. Nel layer successivo gli antigeni ricercano messaggi di eliminazione con il proprio identificatore, se ne trovano terminano il thread. La medesima procedura viene applicata per i linfociti B: gli antigeni rimanenti trasmettono nuovamente la propria posizione (ripetutamente, così da sovrascrivere i messaggi degli antigeni già eliminati) e i linfociti cercano antigeni nel raggio di interazione, l'eventuale legame tra linfocita e antigene è determinato mediante un generatore di numeri casuali, il cui esito viene confrontato con la probabilità di legame dei rispettivi recettori; in caso di legame, l'antigene viene marcato come processato e il linfocita B assume lo stato "attivo". A questo punto, i linfociti B attivi che non hanno ancora interagito nel timestep corrente, inviano posizione e identificatore in attesa di ricevere uno stimolo; i linfociti T cercano linfociti B attivi nelle vicinanze da poter stimolare e, se ne trovano, inviano un segnale di stimolo con l'identificatore del linfocita B interessato. I linfociti B che ricevono il segnale di stimolo, cambiano stato a "stimolato" e producono un predeterminato numero di anticorpi nelle loro vicinanze, in seguito affrontano un processo di ipermutazione, dove modificano il proprio recettore secondo regole probabilistiche. Infine, ogni entità si sposta seguendo le leggi del moto

browniano, aggiustando la posizione finale all'interno dei confini dell'ambiente di simulazione.

Con lo scopo di modellare i comportamenti precedentemente descritti, sono state definite le seguenti agent function:

- *antigen_send_position*: implementa l'invio della posizione degli antigeni, non ha nessun tipo di messaggio in input, mentre in output invia il messaggio *antigen_pos*: un messaggio contenente l'identificatore dell'antigene e l'epitope.

- *antibody_interaction*: implementa l'eliminazione degli antigeni da parte degli anticorpi, il messaggio in input è antigen_pos, mentre in output invia il messaggio *antigen_kill*: un messaggio che riporta solamente l'identificatore dell'antigene.

- *antigen_receive_kill*: applica l'eliminazione effettiva, il messaggio in input è antigen_kill e non c'è alcun output.

- *b_cell_binding*: implementa il tentativo dei linfociti B di legarsi agli antigeni, il messaggio in input è antigen_pos e l'eventuale messaggio in output è antigen_kill.

- *b_cell_send_position*: implementa l'invio della posizione dei linfociti B, non ha nessun tipo di messaggio in input, mentre in output invia il messaggio *b_cell_pos*: un messaggio contenente l'identificatore del linfocita.

- *t_cell_send_stimulus*: implementa il tentativo dei linfociti T di stimolare i linfociti B presentanti l'antigene, il messaggio in input è b_cell_pos e il messaggio in output è *tcell_stimulate*: un messaggio contenente l'identificatore del linfocita.

- *b_cell_receive_stimulus*: implementa la ricezione dello stimolo da parte dei linfociti B, mutandone lo stato; il messaggio in input è t_cell_stimulate, non c'è output.

- *spawn_antibodies*: implementa la generazione di anticorpi nei pressi dei linfociti B, la quantità di anticorpi è fissa; questa agent function non fa uso di messaggi.

- *duplication*: implementa la duplicazione dei linfociti B, avviene dopo la produzione degli anticorpi; non fa uso di messaggi.

- *diffuse_entities*: implementa la diffusione di cellule e molecole all'interno del sistema, questa agent function non fa uso di messaggi.

4.1.4 Host Function

Le host function definite nel modello sono le seguenti:

- *init_agents*: si tratta della init function, al suo interno vengono inizializzati gli agenti secondo i valori definiti nell'environment.

- *plot_agents*: è una host function che si occupa della generazione di un grafico utilizzando la libreria `matplotlibcpp`, esegue solo ad ogni 10% di svolgimento della simulazione.

4.2 Architettura del codice

4.2.1 Struttura tipica di un progetto FLAME GPU

L'architettura del progetto è ispirata ai modelli agent-based proposti dal team di sviluppo di FLAME GPU [5, 6]. La configurazione della simulazione è totalmente situata file `main.cu`, il punto di ingresso del programma. il codice è suddiviso in:

1. Definizione delle agent/host/init function.

2. Inizializzazione dell'environment.

3. Creazione degli agenti.

4. Definizione dei messaggi.

5. Assegnazione delle agent function agli agenti.

6. Creazione dei layer e distribuzione delle funzioni sui layer.

7. Avvio della simulazione.

4.2.2 Definizione delle Funzioni

La definizione di funzioni su modelli FLAME GPU avviene direttamente nel file `main.cu`; questo perché non sono funzioni normali, utilizzano delle *keyword* specifiche fornite dalla libreria di FLAME GPU: `FLAMEGPU_AGENT_FUNCTION`, `FLAMEGPU_HOST_FUNCTION` e `FLAME_GPU_INIT_FUNCTION`. L'utilizzo di queste keyword impedisce la dichiarazione anticipata delle funzioni agente, che pertanto vanno definite immediatamente ed è impossibile definirle in file esterni a quello in cui vengono direttamente utilizzate.

Segue un esempio di definizione di una agent function:

```
1 FLAMEGPU_AGENT_FUNCTION(antibody_interaction, flamegpu::
     MessageSpatial2D, flamegpu::MessageSpatial2D) {
2  ...
3 }
```

La signature della funzione comprende: il nome, il messaggio in input e il messaggio in output; al momento della dichiarazione non è noto il messaggio utilizzato, si specifica solamente la strategia di comunicazione che si desidera adottare, se non si prevede l'utilizzo di messaggi si usa la parola chiave `flamegpu::MessageNone`

All'interno di un agent function si può accedere ai dati dell'agente che la esegue tramite la funzione `FLAMEGPU->getVariable<T>()`, che restituisce un dato di tipo T, fornendo come parametro il nome della variabile dell'agente.

```
1 const float x = FLAMEGPU->getVariable<float>("x");
```

In questo caso, se l'agente ha una variabile "x" di tipo float, viene restituito il suo valore attuale, altrimenti viene lanciata un eccezione. È anche possibile modificare il valore delle variabili all'interno delle agent function con l'analogo metodo `FLAMEGPU->setVariable<T>()`, passando come parametri il nome della variabile e il nuovo valore che si vuole assegnare.

Si possono leggere le costanti salvate nell'environment tramite la funzione `FLAMEGPU->environment.getProperty<T>()`, passando come argomento il nome della proprietà.

```
1 x2 < FLAMEGPU->environment.getProperty<float>("width")
```

Per quanto riguarda la lettura e la scrittura dei messaggi si usano:

- `FLAMEGPU->message_in`: restituisce un iterabile di messaggi ricevuti in base alla strategia di comunicazione, per i messaggi Spatial2D è necessario passare i parametri x e y, ovvero la posizione dell'agente; ogni messaggio ha delle variabili alle quali si può accedere tramite `getVariable<T>()`.

- `FLAMEGPU->message_out`: rappresenta un messaggio le cui variabili sono ancora da definire, espone solamente la funzione `setVariable<T>()` per modificarle.

```
1 for (auto entity : FLAMEGPU->message_in(x, y)) {
2     ...
3     FLAMEGPU->message_out.setVariable<float>("x", entity
          .getVariable<float>("x"));
4     ...
5 }
```

Tuttavia è necessario far notare dei dettagli implementativi riguardo all'uso della strategia di comunicazione Spatial2D, in quanto non banali. Questa strategia di comunicazione opera secondo un raggio `RADIUS`, entro il quale recupera i messaggi inviati dagli agenti. Il raggio dovrebbe descrivere un area circolare entro la quale i messaggi vengono considerati, ma la documentazione ufficiale FLAME GPU (vedi la sezione Agent Functions e le relative sottosezioni Agent Communication, Reading Messages e Spatial Messaging in [4]) specifica che è necessario controllare direttamente le coordinate dei messaggi che si ricevono, in quanto è possibile che ricadano al di fuori del raggio; questo perché il caricamento dei messaggi avviene a caselle (o *tiles*) e non ricopre precisamente una circonferenza.

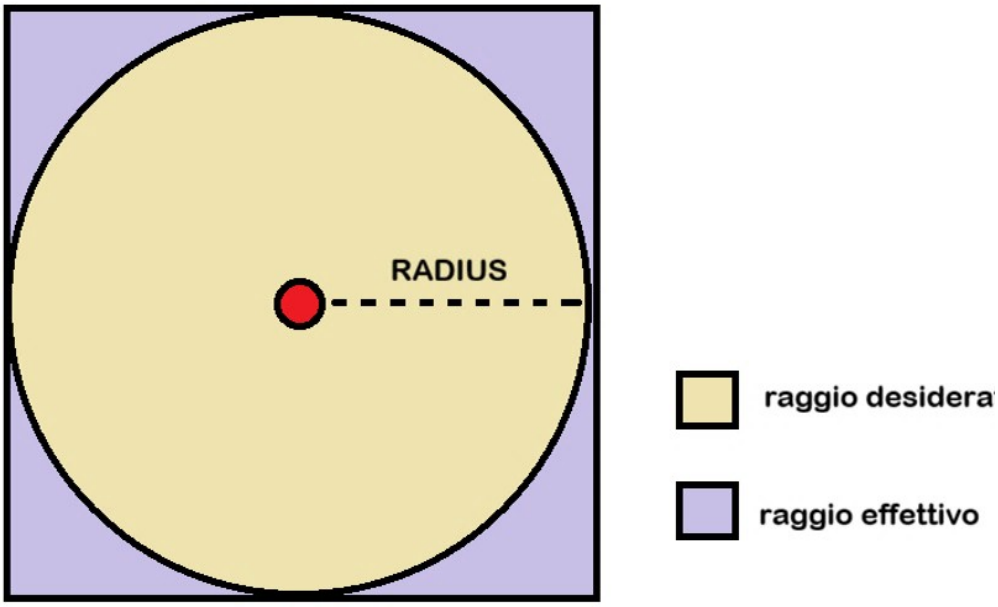

Figura 4.1: Raggio di caricamento dei messaggi Spatial2D su FLAMEGPU2

Il controllo effettuato avviene verificando che il modulo del messaggio, calcolato relativamente alla posizione dell'agente, sia inferiore o uguale a `RADIUS` per ogni messaggio.

```
for (auto entity : FLAMEGPU->message_in(x, y)) {
    const float x2 = entity.getVariable<float>("x");
    const float y2 = entity.getVariable<float>("y");
    ...
    // check if antigen is in range for interaction.
    float x21 = x2 - x;
    float y21 = y2 - y;
    const float separation = sqrt( x21*x21 + y21*y21 );
    if (separation < RADIUS && separation > 0.0f) {
        ...
    }
}
```

Ogni agent function può terminare con due esiti possibili:

1. `FLAMEGPU::ALIVE`: esegue l'equivalente di una classica istruzione `return`: l'agente esce dalla funzione e continua a svolgere le sue normali attività.

2. `FLAMEGPU::DEAD`: l'agente termina l'esecuzione.

È necessario che tutte le funzioni siano definite prima della definizione della modello.

4.2.3 Definizione del Modello

Nel corpo della funzione `main` avviene la definizione del modello tramite la creazione di un oggetto `ModelDescription`; l'interfaccia esposta da questo oggetto espone i metodi necessari per configurare l'environment, gli agenti e le funzioni.

```
// Model creation
flamegpu::ModelDescription model("ImmSimModel");
```

Inizializzazione dell'Environment

L'environment viene modificato tramite l'oggetto `EnvironmentDescription`; questo oggetto non viene istanziato a runtime, bensì viene restituito dall'oggetto `ModelDescription`.

```
flamegpu::EnvironmentDescription env = model.Environment
    ();
```

L'aggiunta di nuove proprietà all'environment avviene tramite il metodo `newProperty<T>()`, dove T è il tipo del dato. Il metodo accetta due parametri: il nome della nuova proprietà e il valore assegnatole.

```
// Grid dimensions.
env.newProperty<float>("width", params.width);
env.newProperty<float>("height", params.height);
```

Definizione degli Agenti

La definizione di nuovi agenti all'interno di un modello FLAME GPU 2 avviene tramite il metodo `newAgent()` esposto dell'interfaccia dell'oggetto `ModelDescription`, questo metodo restituisce un oggetto `AgentDescription`.

```
// B-Lymphocyte agent.
flamegpu::AgentDescription bLymph = model.newAgent("B-
    Lymphocyte");
```

È possibile aggiungere variabili a un oggetto `AgentDescription` tramite il metodo `newVariable<T>()`, dove T è il tipo della variabile. I parametri accettati sono: il nome e, opzionalmente, un valore di default.

```
bLymph.newVariable<float>("x");
bLymph.newVariable<float>("vx", 0.0f);
```

Gli agenti possono anche contenere varibili vettoriali, si fa uso del metodo `newVariable<T, n>()` dove `T` è il tipo di ogni elemento dell'array e `n` è il numero di elementi dell'array. I parametri accettati: sono il nome della variabile e, opzionalmente, gli elementi.

```
bLymph.newVariable<unsigned char, RECEPTOR_SIZE>("
    receptor");
```

Ogni agente ha una variabile dichiarata implicitamente: l'`ID`; un intero senza segno mascherato dal tipo `flamegpu::id_t` diverso per ogni istanza di agente nella simulazione, si ottiene tramite la funzione `FLAMEGPU->getID()`.

FLAME GPU 2 supporta solo determinati tipi di variabili per gli agenti:

- **char**
- **int8_t** (signed char)
- **uint8_t** (unsigned char)
- **int16_t** (signed short)
- **uint16_t** (unsigned short)
- **int32_t** (int)
- **uint32_t** (unsigned int)

- **int64_t** (long)
- **uint64_t** (unsigned long)
- **float**
- **double**
- **flamegpu::id_t**

Per ulteriori dettagli si può fare riferimento alla sezione Creating a Model, Supported Types in [4].

Definizione dei Messaggi

La definizione dei messaggi ottiene soddisfacendo due requisiti: scegliere una strategia di comunicazione e determinare le variabili contenute dal messaggio. Anche la definizione dei messaggi avviene tramite la chiamata dei metodi di interfaccia di un oggetto "Description", con una differenza: la scelta della strategia coincide con la creazione dell'istanza dell'oggetto.
Il metodo `newMessage<T>()` dell'oggetto `ModelDescription` crea un nuovo tipo di messaggio; `T` è la strategia di comunicazione che si vuole utilizzare con il nuovo messaggio, l'unico parametro accettato è il nome del nuovo tipo di messaggio.

```
1 // Antigen position message
2 flamegpu::MessageSpatial2D::Description antigen_pos =
    model.newMessage<flamegpu::MessageSpatial2D>("
    antigen_pos");
```

Alcune strategie di comunicazione richiedono la presenza di determinate variabili all'interno dei messaggi; tali variabili sono, dunque, definite implicitamente e sono necessarie alla corretta ricezione dei messaggi da parte di ciascun agente. Ad esempio, la strategia Spatial2D include implicitamente l'uso delle variabili `x` e `y`.
La strategia Spatial2D richiede anche la scelta di parametri come:

- **Minimo**: per minimo, si intende la coppia di valori x e y che costituisce il limite inferiore per ogni altra coppia di coordinate all'interno della simulazione. Si impostano tramite il metodo `setMin(minX, minY)`.
- **Massimo**: per massimo, si intende la coppia di valori x e y che costituisce il limite inferiore per ogni altra coppia di coordinate all'interno della simulazione. Si impostano tramite il metodo `setMax(maxX, maxY)`.

- **Raggio**: la distanza entro la quale sono visibili i messaggi; si imposta con tramite il metodo `setRadius(r)`.

```
antigen_pos.setMin(0.0f, 0.0f);
antigen_pos.setMax(env.getProperty<float>("width"), env.
    getProperty<float>("height"));
antigen_pos.setRadius(env.getProperty<float>("
    interaction_radius"));
```

Per aggiungere variabili ad un nuovo tipo di messaggio si usa la medesima procedura degli agenti: tramite il metodo `newVariable<T>()`.

```
antigen_pos.newVariable<flamegpu::id_t>("id");
antigen_pos.newVariable<unsigned char, RECEPTOR_SIZE>("
    receptor");
```

Assegnazione delle Agent Function agli Agenti

L'assegnazione (o *binding*) delle agent function agli agenti avviene tramite l'interazione tra due oggetti: `AgentDescription` e `AgentFunctionDescription`; per ottenere un oggetto `AgentFunctionDescription` si chiama, da `AgentDescription`, il metodo `newFunction()`. Tale metodo accetta come parametri: il nome della agent function che vogliamo creare e il nome dichiarato nella signature della definizione della stessa agent function.

```
auto a_i = antibody.newFunction("antibody_interaction",
    antibody_interaction);
```

Dopodiché, è necessario mappare correttamente i messaggi all'agent function; per farlo si usano i due metodi: `setMessageInput()` e `setMessageOutput()`, entrambi esposti dall'oggetto `AgentFunctionDescription`. Entrambi i metodi accettano un solo parametro: il nome del tipo di messaggio che si vuole utilizzare, dichiarato in precedenza.

```
a_i.setMessageInput("antigen_pos");
a_i.setMessageOutput("antigen_kill");
```

Nel caso in cui una agent function possa portare alla terminazione di thread agenti o alla generazione di nuovi thread agenti è necessario configurare le rispettive variabili all'assegnamento:

- **Allow agent death**: è un flag che può assumere i valori true e false, se impostato a true la funzione può terminare con lo stato `FLAMEGPU::DEAD`.
- **Agent output** è una variabile a cui è necessario assegnare un nome di agente definito in precedenza, se assegnato la funzione potrà opzionalmente istanziare agenti.

È possibile impostare queste variabili tramite due metodi esposti dall'interfaccia dell'oggetto: `setAgentOutput()` e `setAllowAgentDeath()`.

```
auto recv_kill = antigen.newFunction("
    antigen_receive_kill", antigen_receive_kill);
recv_kill.setMessageInput("antigen_kill");
// Allows agent death inside this function.
recv_kill.setAllowAgentDeath(true);
 ...
auto spawn = bLymph.newFunction("spawn_antibodies",
    spawn_antibodies);
spawn.setAgentOutput("Antibody");
```

L'agente in output viene istanziato con i valori di default, ma è sempre possibile modificarli tramite il metodo `setVariable<T>()` esposti dall'oggetto `FLAMEGPU->agent_out` accessibile all'interno dell'agent function. Un agente può istanziare un solo agent per agent function.

Creazione dei Layer

La creazione dei layer avviene tramite la configurazione di oggetti `LayerDescription`, generati dal metodo `newLayer` esposto da `model`. L'ordine di esecuzione dei layer è l'ordine in cui essi vengono definiti.
Ogni layer contiene delle agent/host function da eseguire. È possibile aggiungere funzioni ai layer tramite i metodi: `addAgentFunction()` e `addHostFunction`. Se si aggiunge un host function, il parametro della funzione è un riferimento alla definizione della funzione; se si aggiunge una agent function è necessario ottenere il riferimento alla `AgentFunctionDescription` tramite il metodo `getFunction()` esposto dall'oggetto `AgentDescription`, come argomento si passa il nome con cui la funzione è stata assegnata all'agente.

```
// 11th Layer
flamegpu::LayerDescription layer11 = model.newLayer("
    Layer␣11");
```

```
layer11.addAgentFunction(bLymph.getFunction("
    spawn_antibodies"));

// 13th Layer
flamegpu::LayerDescription layer13 = model.newLayer("
    Layer 13");
layer13.addAgentFunction(bLymph.getFunction("
    diffuse_entities"));
layer13.addAgentFunction(tLymph.getFunction("
    diffuse_entities"));
layer13.addAgentFunction(antibody.getFunction("
    diffuse_entities"));
layer13.addAgentFunction(antigen.getFunction("
    diffuse_entities"));

// 14th Layer
flamegpu::LayerDescription layer14 = model.newLayer("
    Layer 14");
layer14.addHostFunction(reinsert_antigens);
```

L'init function non fa parte di un layer specifico, viene aggiunta direttamente alla `ModelDescription` tramite il metodo `addInitFunction()`.

```
// Set the initialization function defined above.
model.addInitFunction(init_agents);
```

Avvio della simulazione

Una volta configurato l'intero modello è necessario creare l'oggetto `flamegpu::CUDASimulation`, che rappresenta la simulazione stessa a partire dal modello definito.

```
flamegpu::CUDASimulation sim(model);
```

Il numero di timestep che la simulazione dovrà eseguire viene assegnato tramite il campo `SimulationConfig().steps`.

```
sim.SimulationConfig().steps = params.steps;
```

La simulazione viene avviata chiamando il metodo `simulate()`.

```
1 sim.simulate();
```

Capitolo 5

Risultati

5.1 Valutazione della Correttezza

Il programma ha prodotto risultati soddisfacenti, simulando correttamente la risposta umorale del sistema immunitario. Le immagini mostrano lo svolgimento di una simulazione di 5000 timestep, con 2000 antigeni, 200 linfociti B e 200 linfociti T; a metà della simulazione vengono reintrodotti 2000 antigeni per verificare il miglioramento della risposta immunitaria dopo la produzione di anticorpi.

Nel grafico iniziale (Figura 5.1) le entità sono disposte casualmente nell'ambiente di simulazione e la produzione di anticorpi non è ancora iniziata. Gli antigeni sono rappresentati in rosso, i linfociti B in verde, i linfociti T in magenta e gli anticorpi in blu.

Dopo 1250 timestep (Figura 5.2), i linfociti B si sono legati agli antigeni, iniziando a produrre anticorpi; questi ultimi hanno eliminato completamente gli antigeni.

A metà della simulazione (Figura 5.3), sono stati reinseriti gli antigeni e gli anticorpi già presenti sono sufficienti ad eliminare tempestivamente gli antigeni, limitando le interazioni tra antigeni e linfociti B.

Nell'ultimo grafico (Figura 5.4) possiamo notare che gli anticorpi non sono aumentati nella stessa proporzione; questo perché gli anticorpi hanno eliminato gli antigeni prima che i linfociti B potessero interagirvi.

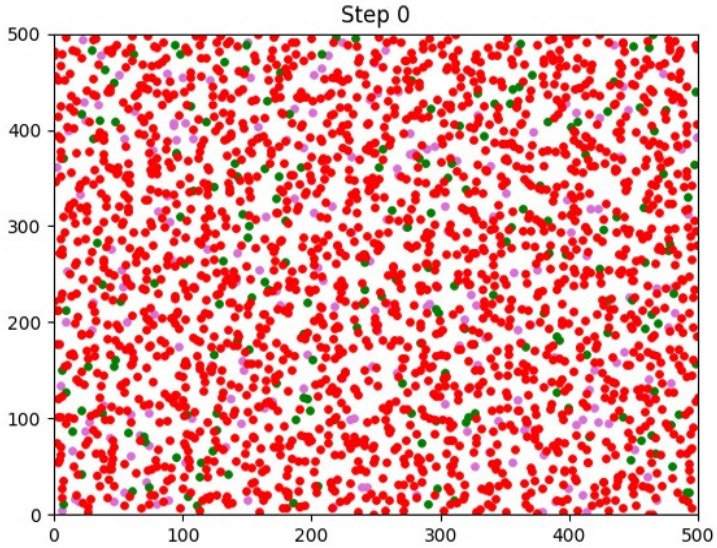

Figura 5.1: Grafico a inizio simulazione

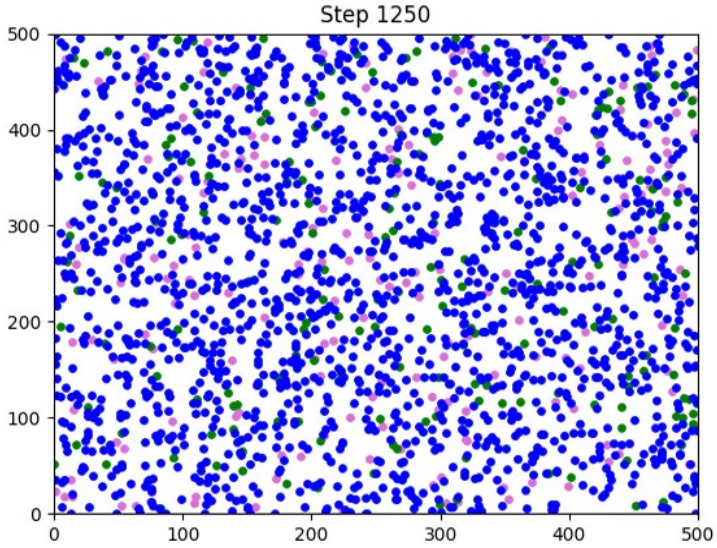

Figura 5.2: Grafico dopo 1250 timestep

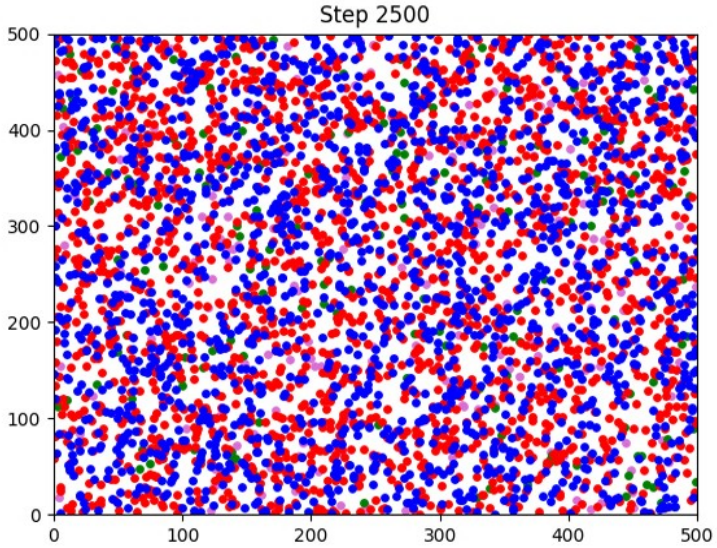

Figura 5.3: Grafico a metà simulazione, dopo 2500 timestep

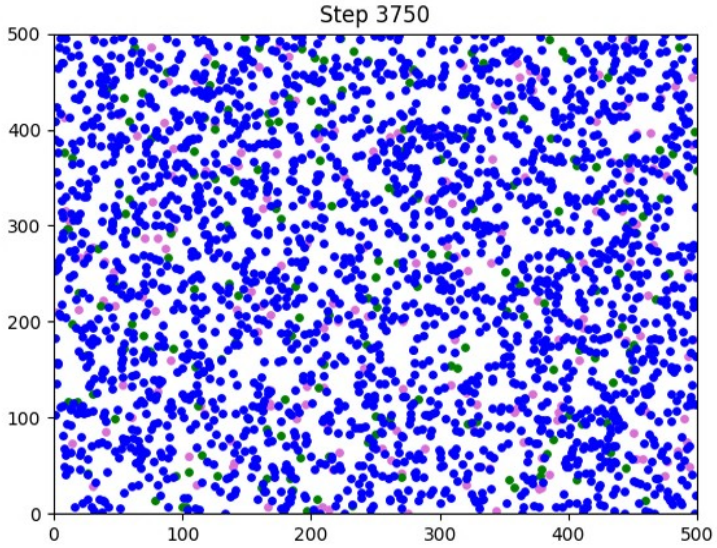

Figura 5.4: Grafico a 3750 timestep

5.2 Valutazione delle Prestazioni

Per confrontare accuratamente le tre versioni (seriale, CUDA [7] e FLAME GPU), verranno effettuati diversi test a parità di parametri, attenendosi al seguente criterio [7, p.35]:

- Il numero di antigeni varierà tra 400 e 2400, spostandosi a intervalli di 400;
- Il numero di linfociti sarà sempre il 10% degli antigeni;
- La dimensione dell'ambiente varierà tra 200 e 1000;
- Il numero di timestep sarà sempre 5000.

I grafici (Figura 5.5) mettono a confronto i tempi di esecuzione delle versioni.

La versione seriale presenta una crescita dei tempi lineare al variare del numero degli antigeni e non lineare al variare delle dimensioni della griglia, in cui si raggiungono risultati ottimali (a parità di antigeni) attorno ai 600.

La versione FLAME GPU ottiene tempi decisamente migliori rispetto alla versione seriale. Si ottiene più consistenza nei risultati con dimensioni medie della griglia, griglie di dimensioni troppo piccole (200) o troppo grandi (1000) producono risultati peggiori; il modello risponde bene allo scalare del numero di agenti.

La versione CUDA è quella che raggiunge i tempi migliori, i tempi di esecuzione crescono in maniera lineare, ma molto lentamente. Questa versione non supporta una dimensione variabile della griglia, pertanto, i risultati visibili nel grafico valgono solo per una griglia di dimensione fissata a 500. Con un modesto numero di agenti, la versione FLAME GPU è pesantemente sfavorita rispetto a CUDA, in quanto necessita di un overhead di inizializzazione. All'aumentare del numero complessivo di agenti (e, di conseguenza, di interazioni), l'overhead inizia a diventare meno rilevante rispetto al carico computazionale; queste condizioni rendono la versione FLAME GPU competitiva, ma non migliore (Figura 5.6).

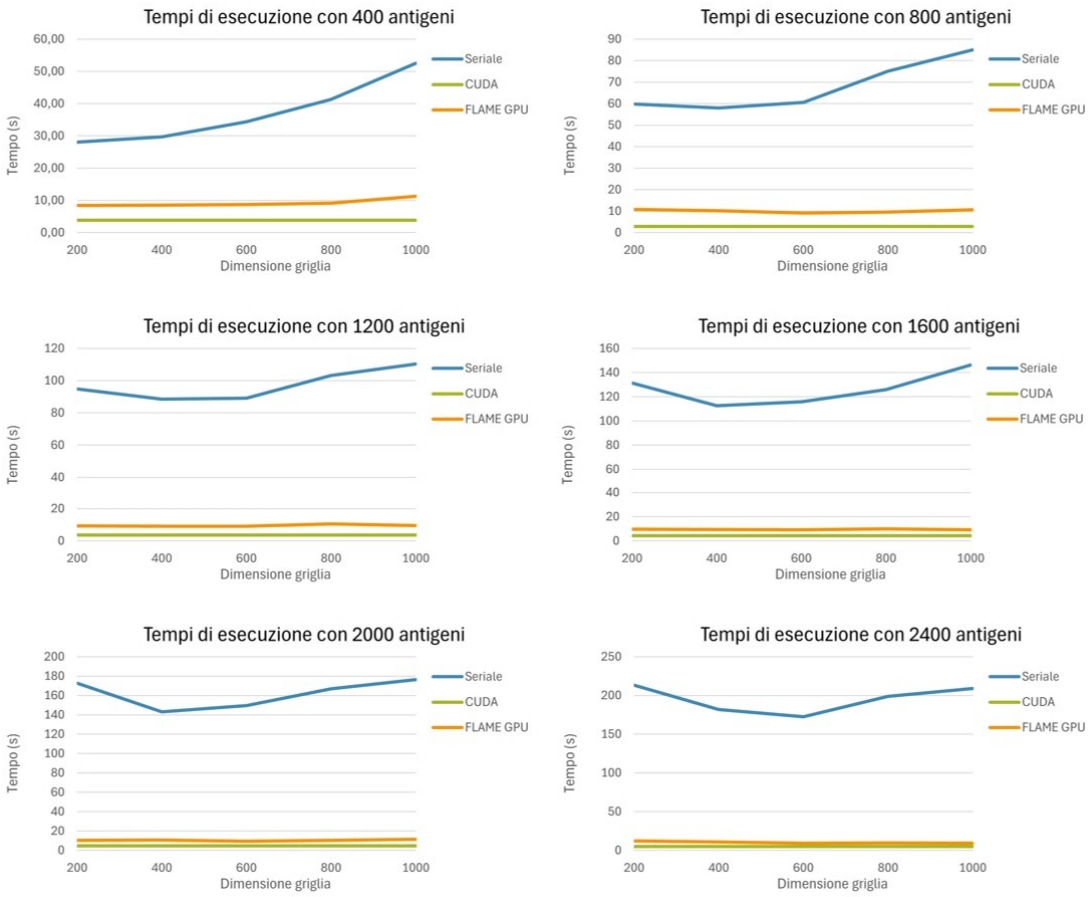

Figura 5.5: Tempi di esecuzione al variare del numero di antigeni.

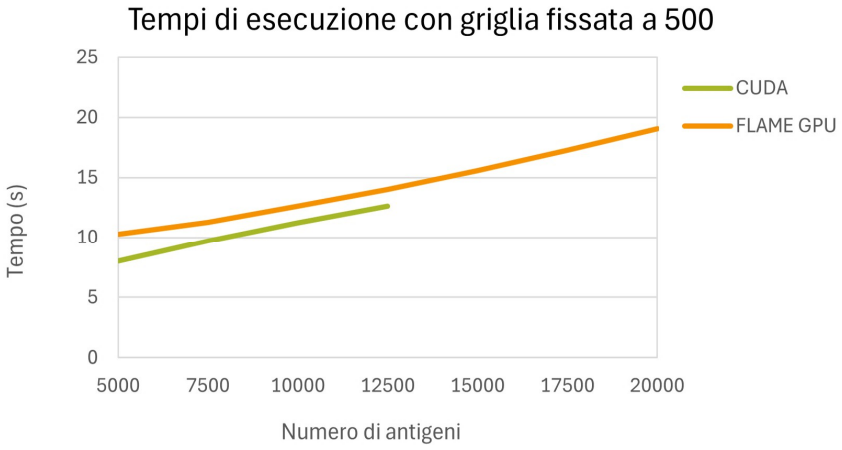

Figura 5.6: Tempi di esecuzione delle due versioni parallele con un elevato numero di agenti.

Non è stato possibile misurare ulteriori risultati della versione CUDA per via di limiti legati all'allocazione di memoria.

Per determinare i miglioramenti apportati dalle versioni parallele, verrà calcolato lo *speedup* a partire dai tempi ottenuti dai test. Lo speedup (S) è definito dal rapporto tra il tempo di esecuzione del programma seriale (T_s) e il tempo di esecuzione del programma parallelo (T_p).

$$S = \frac{T_s}{T_p} \tag{5.1}$$

I risultati mostrano un chiaro trade-off tra prestazioni e complessità di sviluppo.
L'implementazione CUDA permette il raggiungimento di uno speedup medio di 24× (il picco raggiunge il 35×) rispetto al programma seriale, a fronte di un implementazione complessa, richiedente la gestione esplicita della memoria GPU, della sincronizzazione dei thread e di una struttura del codice estremamente più articolata.

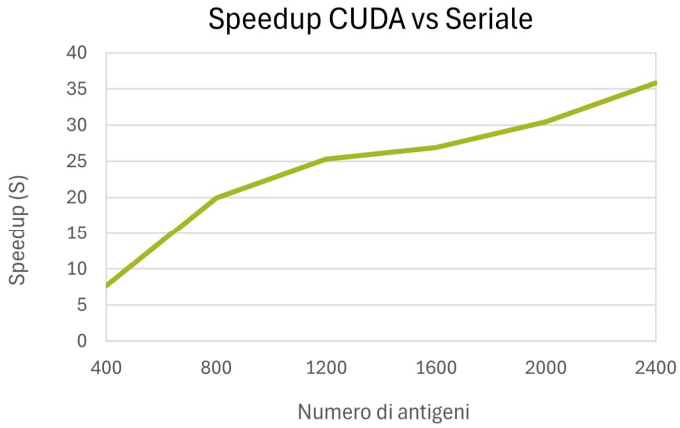

Figura 5.7: Speedup apportato dalla versione CUDA.

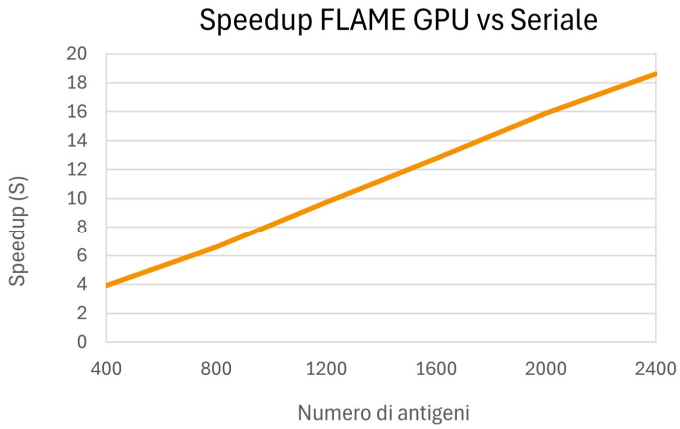

Figura 5.8: Speedup apportato dalla versione FLAME GPU.

Al contrario, l'implementazione FLAME GPU ottiene uno speedup minore, di media 11× (con picco a 18×), ma riduce drasticamente la complessità di implementazione del modello. Tale riduzione si manifesta sia in termini di dimensione del codice (Circa metà delle righe di codice rispetto alla versione CUDA), sia a livello di astrazione, demandando al framework la gestione di aspetti come memoria, sincronizzazione e concorrenza.

La scelta tra i due approcci, dunque, non dipende esclusivamente dalle prestazioni assolute, bensì dal contesto applicativo e dagli obiettivi del progetto. Un progetto che prevede frequenti estensioni del dominio applicativo tende a privilegiare produttività e manutenibilità del software, accettando un compromesso in termini di efficienza, mentre un progetto che non prevede variazioni del dominio applicativo può orientarsi verso soluzioni maggiormente specializzate, volte a massimizzare l'efficienza computazionale.

Conclusioni

La tesi ha raggiunto gli obiettivi prefissati: è stata realizzata un'implementazione del modello Celada-Seiden tramite il framework di simulazione ad agenti FLAME GPU.

È stata fornita una descrizione del modello teorico, in cui sono state definite le principali funzioni e agenti del sistema immunitario. A partire dal modello teorico, è stata illustrata la progettazione e, successivamente, l'implementazione derivante da essa.

Il programma realizzato riproduce correttamente il comportamento dell'implementazione utilizzata come riferimento iniziale. Le prestazioni sono buone, raggiungendo tempi di esecuzione circa 11 minori rispetto alla versione seriale.

Il lavoro svolto in questa tesi offre spunti per eventuali sviluppi futuri. Grazie alla duttilità implementativa offerta da FLAME GPU, è possibile espandere ulteriormente le dinamiche di interazione implementate:

- Introduzione dei complessi immuni come agente concreto.
- Implementare i recettori per tutte le entità del modello, al momento esistono solo per l'antigene e i linfociti B.
- Implementare il riconoscimento tramite MHC: le cellule che assimilano antigeni dovrebbero trasportarne i frammenti (detti peptidi) e, successivamente, i linfociti T innescano la produzione di anticorpi in base ai peptidi riconosciuti come estranei.

Si può anche valutare l'implementazione di modelli immunologici più recenti e complessi.

Il codice sorgente realizzato per questa implementazione può essere consultato dal seguente repository:
https://github.com/samuHouse/ImmSim-FLAMEGPU2-model

Bibliografia

[1] Franco Celada e Philip E. Seiden. “A computer model of cellular interactions in the immune system”. In: *Immunology Today* (1992).

[2] FLAME GPU Developers. *FLAME GPU*. Ultimo accesso: 10 gennaio 2026. 2020. URL: https://flamegpu.com/.

[3] FLAME GPU Developers. *FLAME GPU Design Philosophy*. Ultimo accesso: 15 dicembre 2025. 2025. URL: https://docs.flamegpu.com/tutorial/index.html.

[4] FLAME GPU Developers. *User Guide*. Ultimo accesso: 14 dicembre 2025. 2025. URL: https://docs.flamegpu.com/guide/index.html.

[5] FLAME GPU Development Team. *FLAME GPU 2 Template for CUDA C++*. https://github.com/FLAMEGPU/FLAMEGPU2-model-template-cpp. Accessed: 2025-01-02. 2020.

[6] FLAME GPU Development Team. *FLAME GPU 2 Tutorial (CUDA C++)*. https://github.com/FLAMEGPU/FLAMEGPU2-tutorial-cudacpp. Accessed: 2025-01-02. 2020.

[7] Maurizio Daniel Pellanda. “Simulazione del Sistema Immunitario Umano tramite Programmazione Parallela”. Tesi di dott. Università di Bologna, 2025. URL: https://amslaurea.unibo.it/id/eprint/34485/.

[8] Paul Richmond et al. *Gerarchia della simulazione GPU*. Ultimo accesso: 14 dicembre 2025. 2023. URL: https://developer.nvidia.com/blog/fast-large-scale-agent-based-simulations-on-nvidia-gpus-with-flame-gpu/.

[9] Wikipedia. *Legge di Moore*. Ultimo accesso: 10 gennaio 2026. 2025. URL: https://it.wikipedia.org/wiki/Legge_di_Moore.