Università degli Studi di Milano - Bicocca
**Scuola di Scienze**
**Dipartimento di Informatica, Sistemistica e Comunicazione**
**Corso di Laurea in Informatica**

# Simulazione stocastica su Graphics Processing Units di modelli di proliferazione cellulare

**Relatore:** *Prof. Daniela Besozzi*

**Co-relatore:** *Dott. Simone Spolaor*

**Relazione della prova finale di:**
*Eric Nisoli*
*Matricola 807147*

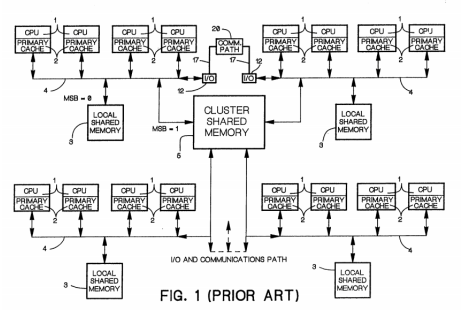
**Anno Accademico 2017-2018**

# Indice

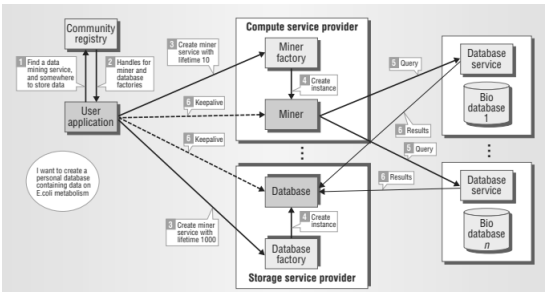
# Capitolo 1

# Introduzione

L'evoluzione di popolazioni di cellule proliferanti risulta spesso estremamente difficile da caratterizzare mediante convenzionali tecniche di biologia sperimentale [3]. Tuttavia, un'analisi approfondita di questi fenomeni faciliterebbe lo studio e la comprensione di diverse patologie umane. Per questa ragione, è necessario accoppiare metodologie computazionali avanzate ai classici esperimenti di laboratorio. Nel caso particolare della Leucemia Mieloide Acuta (LMA) si pensa che l'eterogeneità tra diverse sottopopolazioni cellulari presenti all'interno del tumore giochi un ruolo fondamentale per quanto riguarda la resistenza alle cure e il fatto che il tumore possa ripresentarsi a seguito del completamento della terapia [5]. Per lo studio di questo fenomeno si può optare per un approccio model-driven in modo da studiare le dinamiche riguardanti la proliferazione di popolazioni cellulari eterogenee [1]. L'obiettivo di questo stage è lo sviluppo di un simulatore di divisione cellulare in grado di tenere conto della presenza di differenti tipi di popolazioni cellulari e della stocasticità degli eventi di divisione ad essi correlati. Le cellule si divono in maniera esponenziale, quindi la simulazione di questi fenomeni comporta costi in termini di memoria e tempo esponenziali, infatti utilizzando un simulatore implementato su Central Processing Unit (CPU) è emerso l'elevato costo computazionale necessario alla simulazione di popolazioni cellulari di grandi dimensioni [1]. A causa di ciò si è reso necessario l'utilizzo di tecniche di accelerazione software per incrementare le performance del simulatore. Esistono diverse metodologie e architetture il cui obiettivo è l'incremento delle prestazioni dei software per cercare di ridurne l'impatto in termini di costo computazionale di risorse sia fisiche, come la disponibilità di spazio di memorizzazione, sia temporali, ovvero il tempo necessario all'esecuzione del software. L'insieme di questi paradigmi costituisce l'High Performance Computing (HPC), solitamente utilizzato in ambito scientifico per ridurre i costi legati a simulazioni di natura biologica, chimica e fisica. Generalmente all'interno di questa categoria possono essere distinti tre diversi tipi di architettura: il Computer cluster, il Grid computing e il General-Purpose computing on Graphics Processing Units (GPGPU). Ognuna di queste architetture propone strategie differenti per quanto riguarda la risoluzione delle problematiche precedentemente elencate.




(A) Schema Computer cluster [7]




(B) Schema Grid computing [6]

FIGURA 1.1: Rappresentazione schematica del funzionamento di due architetture di HPC: Computer cluster e Grid computing

Nei Computer cluster le elaborazioni sono svolte da CPU singole (denominate nodi) e distribuite su macchine di elaborazione fisicamente differenti, come si può evincere dalla Figura 1.1 (a destra) e utilizzano una rete locale per comunicare tra di loro. Questi cluster generalmente sono collocati all'interno dello stesso luogo (una stanza o un intero edificio), risentono dunque solamente del ritardo generato da una rete locale di computer. Solitamente questi cluster vengono utilizzati per simulazioni che necessitano di enormi quantità di dati per essere svolte, dunque per la simulazione della proliferazione cellulare sarebbe uno spreco di risorse di elaborazione visto che i dati di input sono dell'ordine dei megabyte (MB).

Il Grid computing, mostrato in Figura 1.1 (a sinistra), prevede che diversi enti ed organizzazioni mettano in condivisione risorse e dati per portare a termine la medesima finalità. I nodi delle risorse (sia hardware che softawre) possono essere dislocati anche geografimente in luoghi differenti; il vantaggio è la disponibilità di grandi quantità di dati e hardware per l'elaborazione degli stessi che non sarebbe possibile ottenere utilizzando un'architettura Cluster. Anche in questo caso l'utilizzo di questa tipologia di elaborazione non è conveniente per la simulazione della proliferazione cellulare, poiché è necessaria solamente la cooperazione con il laboratorio di analisi che fornisce i dati necessari.

Per l'implementazione della nuova versione del simulatore è stato scelto di sfruttare il paradigma del GPGPU computing, in quanto necessita solamente di una Graphics Processing Unit (GPU) installata all'interno di un computer. Il vantaggio è l'immediata disponibilità della macchina quando si devono effettuare delle simulazioni, dato che non è necessario fare richiesta preventiva di allocazione di risorse agli enti che gestiscono un Computer cluster oppure utilizzare il paradigma del Grid computing. L'implementazione del GPGPU computing per lo sviluppo del simulatore è stata resa possibile grazie alla tecnologia Computed Unified Device Architecture (CUDA) che offre la possibilità di sfruttare il parallelismo offerto dalle GPU moderne per ottenere incrementi significativi delle performance delle simulazioni effettuate su modelli di proliferazione cellulare di LMA. Durante lo sviluppo del simulatore è stato utilizzato il parallelismo dinamico per simulare i fenomeni di divisione cellulare inerentemente ricorsivi, trovando immediato riscontro positivo verificabile grazie al significativo incremento delle performance del simulatore. Il confronto dei

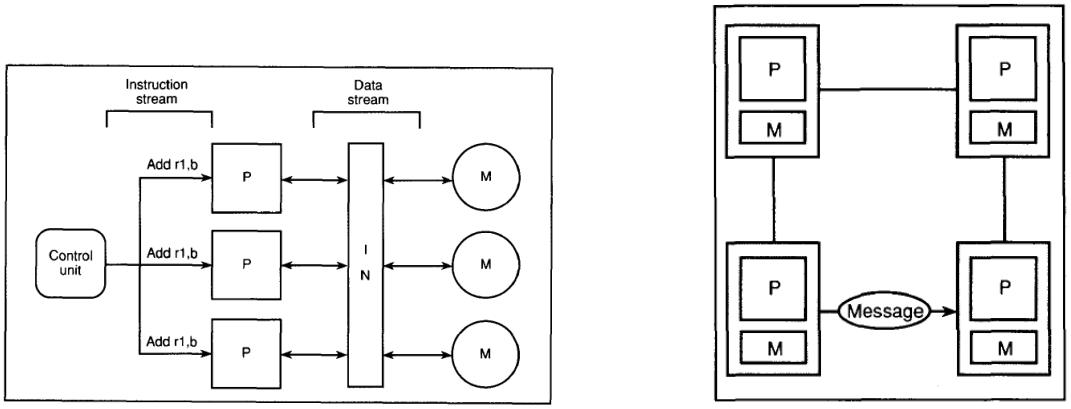
dati ottenuti dalle simulazioni effettuate tramite simulatore per CPU e dalla corrispondente versione parallela sviluppata durante il periodo di stage hanno confermato la correttezza dei risultati ottenibili utilizzando la versione del software implementata con l'ausilio della libreria CUDA.

# Capitolo 2

# Metodi

## 2.1 GPGPU Computing

Originariamente le GPU sono state concepite come co-processori il cui scopo era puramente l'elaborazione grafica, mentre le altre funzionalità non strettamente legate a questo ambito erano gestite dalla CPU. In seguito si è diffuso l'utilizzo del parallelismo fornito dalle GPU per accelerare diversi tipi di software, dando luogo al GPGPU computing, cioè l'utilizzo di questo tipo di hardware per applicazioni non legate all'elaborazione grafica [11]. Sebbene le GPU operino a frequenze molto inferiori rispetto alle CPU, il loro punto di forza risiede nella presenza di molte più unità di elaborazione (denominate core), in grado quindi di eseguire centinaia o migliaia di thread in parallelo rispetto all'esiguo numero disponibile per la CPU. La tipologia di thread istanziabile dalla GPU è ottimizzata per garantire un basso overhead per quanto riguarda la loro creazione e uno scambio di contesto quasi istantaneo, evitando così una diminuzione delle presentazioni dovuta al cospicuo numero di thread generalmente sfruttati per risolvere un determinato task [9].
In generale, ad uno stream di dati di input corrisponde la creazione di un numero finito di thread per elaborare le informazioni ricevute, e solitamente ogni thread esegue le medesime istruzioni applicate ad un sottoinsieme dei dati. Un'architettura che opera secondo questo concetto è stata classificata da Michael J. Flynn con il termine di Single Instruction Multiple Data (SIMD)[4] e prevede la presenza di un'unità centrale di controllo (in questo caso la GPU) che distribuisce le istruzioni da eseguire alle unità di elaborazione (i thread). Queste unità possono accedere alle informazioni attraverso una comunicazione del tipo processo-processo oppure processo-memoria come nel caso delle GPU.
In ambito scientifico l'architettura SIMD è stata utilizzata per la simulazione di fenomeni fisici, chimici e biologici oltre che all'elaborazione delle immagini, grazie al fatto che le singole unità di elaborazione svolgono operazioni anche in virgola mobile con indirizzamento sia a 32 che a 64




(A) Architettura SIMD (B) Architettura MIMD

FIGURA 2.1: Schemi di esecuzione di architetture SIMD e MIMD

bit.

Oltre a SIMD è stata definita anche un'architettura che prevede l'esecuzione di diverse istruzioni su diversi stream di dati, denominata Multiple Instructions Multiple Data (MIMD)[4]. Questa architettura viene utilizzata solamente quando le unità di elaborazione devono eseguire istruzioni eterogenee, dunque è necessaria la presenza di diverse unità di controllo hardware. Essendo la decentralizzazione un punto cardine di questo paradigma, è necessario introdurre anche il concetto di sincronizzazione fra processi, utilizzando lo scambio di messaggi o la memoria condivisa, dato che le macchine operano in modo asincrono. Sebbene sia possibile da parte di un'architettura MIMD emulare il modello SIMD eseguendo il medesimo set di istruzioni per ogni processore $P$, il principale vantaggio di SIMD in ambito GPGPU computing è il collegamento diretto all'interno della stessa macchina tra le GPU impiegate per l'elaborazione e la CPU. Questa è una caratteristica che differisce dai tipi di architetture MIMD più diffusi, come il Computer cluster e il Grid Computing. La Figura 2.1 mostra i due diversi schemi di esecuzione delle architetture definite in precedenza, dove con $P$ è indicato il processore e con $M$ la memoria. Da notare il fatto che, come spiegato precedentemente, lo schema SIMD prevede che l'unità di controllo distribuisca ai processori $P$ lo stream di istruzioni da eseguire elaborando le informazioni provenienti dallo stream di dati, utilizzando anche la memoria $M$ come supporto per lettura e scrittura di informazioni. Al contrario, nello schema MIMD si nota il diverso approccio adottato rispetto a SIMD, ovvero l'assenza di un'unità di controllo centrale e ogni processore $P$ che esegue uno stream di istruzioni differente l'uno dall'altro, con possibilità di accesso solamente alla propria memoria $M$ dedicata e con sincronizzazione e scambio di informazioni tra processori mediante lo scambio di messaggi, sottolineando maggiormente la natura asincrona di questo modello.

Sebbene le GPU aumentino le performance in maniera considerevole, questi vantaggi si ottengono nel momento in cui lo stream di dati assume una dimensione tale per cui l'overhead di creazione dei thread risulti irrisorio rispetto alla computazione che si deve andare ad effettuare. Ad esempio,

nel momento in cui vogliamo effettuare la ricerca di un valore specifico all'interno di un array, se la dimensione dell'input è di poche migliaia di elementi l'utilizzo della GPU risulta inutile poiché le moderne CPU possiedono frequenze elevate che favoriscono l'esecuzione di istruzioni sequenziali su insiemi di pochi elementi, mentre l'istanziamento di un basso numero di thread unito a una frequenza delle unità di elaborazione della GPU molto inferiore rispetto a quella delle CPU non favorisce un approccio parallelo alla risoluzione di questo tipo di problemi.

## 2.2 Architettura CUDA

CUDA è l'architettura di elaborazione parallela, basata sul paradigma SIMD, progettata e sviluppata da NVIDIA che sfrutta la potenza di calcolo delle GPU per aumentare le prestazioni di alcuni tipi di software. L'elaborazione sta lentamente migrando verso il paradigma di co-processing su CPU e GPU, identificati rispettivamente come *host* e *device*, il quale prevede che l'esecuzione della gran parte del carico computazionale venga demandata alla GPU e i risultati presi nuovamente in carico dalla CPU.
Questo nuovo tipo di paradigma ha trovato immediato seguito nel settore della ricerca scientifica, dato che ha contribuito in particolare alla nascita e al miglioramento di software per la simulazione di fenomeni fisici e biologici.

**Specifiche Hardware**
Generalmente l'hardware può cambiare con l'avvento di nuove generazioni di GPU ma la struttura generale si basa sempre sul concetto di Streaming Multiprocessors (SM) [11].
In Figura 2.2 è rappresentata schematicamente l'architettura di uno SM di una GPU NVIDIA della generazione Fermi. Da notare il numero elevato di core disponibili per l'elaborazione delle istruzioni e la presenza di due tipi distinti di memorie che risiedono all'interno dello SM: i registri e la memoria condivisa.

**Astrazione Software**
Dato che vengono periodicamente rilasciate nuove versione dell'architettura hardware delle schede, NVIDIA ha sviluppato delle API per dialogare con la GPU che sono indipendenti dall'architettura del device utilizzato, rendendo così possibile lo sviluppo di software portabile su tutte le schede che supportano CUDA. In generale, la maggior parte di queste API è accessibile attraverso tutte le GPU NVIDIA, altre però richiedono una determinata versione di "compute capability" ovvero un indicatore che riassume le specifiche generali del device e le sue caratteristiche, come ad esempio il parallelismo dinamico che è disponibile solamente su schede con compute capability maggiore di 3.5.
Il modello astratto definisce tre tipologie di oggetti:

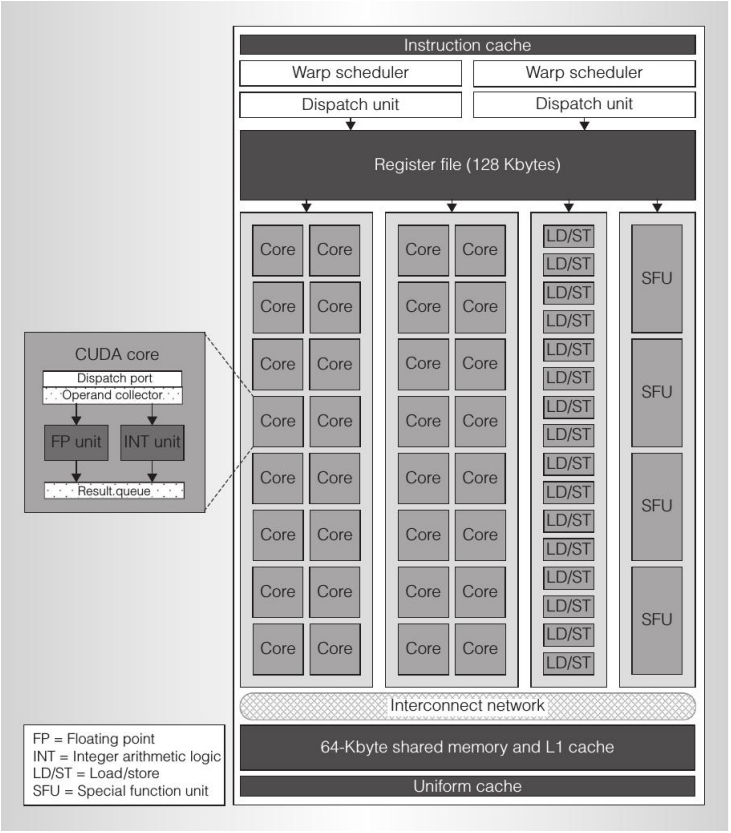



FIGURA 2.2: Streaming Multiprocessor dell'architettura Fermi [11]

- thread: singole unità di calcolo, eseguono il codice sorgente;
- thread block: insieme logico di thread. I thread appartenenti allo stesso thread block hanno accesso ad un'area di memoria condivisa e accessibile solamente da questi ultimi (oltre alla memoria globale della GPU). È inoltre possibile ottenere la sincronizzazione di tutti i thread appartenenti al medesimo thread block;
- grid: insieme logico di thread block. Non è stata prevista un'area di memoria condivisa da tutta la grid e fino ad ora non esiste una primitiva per la sincronizzazione fra thread block di una specifica grid. Per ottenere questo livello di sincronizzazione è necessario invocare la primitiva *cudaDeviceSynchronize()*, operazione che deve essere effettuata dalla CPU. Essendo una funzione sincrona, essa blocca il flusso di esecuzione delle istruzioni della CPU, ponendola in stato di attesa fino al termine dei kernel in esecuzione sulla GPU. Risulta quindi buona norma minimizzare l'utilizzo di questa primitiva, per evitare rallentamenti significativi all'interno del software.

In Figura 2.3 è rappresentato uno schema delle relazioni che intercorrono fra thread, thread block, grid e le tipologie di memorie alle quali essi hanno accesso.
Le procedure che vengono eseguite sulla GPU vengono chiamate kernel (identificabili grazie al prefisso *__global__*) ed è possibile specificarne la dimensione, ovvero decidere quanti thread e

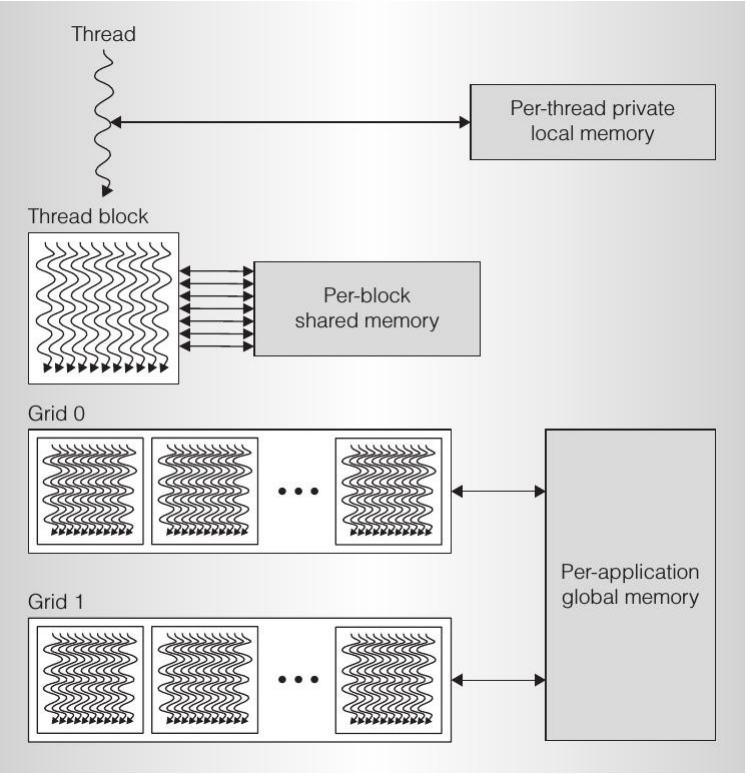



FIGURA 2.3: Schema della gerarchia di thread, thread block, e grid [11]

```
__global__ void my_kernel()
{
    // codice GPU...
}

// CPU code
int main()
{
    int n_blocks = 8; // Numero di thread block
    int n_threads_per_block = 32; // Numero di thread per blocco

    // Esecuzione kernel
    my_kernel<<<n_blocks, n_threads_per_block>>>();
}
```

LISTING 2.1: Esempio di invocazione GPU kernel

thread block verranno assegnati all'esecuzione del codice invocato.

Come è possibile notare nel Listato 2.1, scritto in linguaggio di programmazione CUDA C, si sta invocando l'esecuzione di un kernel specificando l'utilizzo di 8 thread block e 32 thread per blocco. In generale, il numero totale di thread che verranno utilizzati per la computazione del kernel è

$$B * T,$$

dove $B$ e $T$ sono rispettivamente il numero di thread block e il numero di thread per blocco che

```
__global__ void my_kernel(int* array)
{
    // Calcol dell'id globale del thread rispetto al kernel
    int id = threadIdx.x + blockIdx.x * blockDim.x;

    // Elaborazione dell'elemento dell'array corrispondente al thread id
    array[id] = 0;
}
```

LISTING 2.2: Esempio di calcolo dell'id globale associato ad un thread durante l'esecuzione di un kernel

si vuole utilizzare.
La possibilità di modificare la suddivisione del kernel in thread block si rivela molto utile nell'elaborazione di strutture dati non particolarmente complesse, come gli array. Infatti è sufficiente invocare un kernel con numero di thread uguale (o maggiore) al numero di elementi dell'array e assegnare la computazione di ogni elemento ad un singolo e specifico thread.
Esiste però un limite al numero di thread per thread block che è possibile dichiarare (nelle nuove versioni di CUDA è 1024), dunque per ottenere l'id globale del thread relativo al kernel eseguito bisogna avvalersi anche del numero di thread block richiesti durante l'invocazione del kernel, secondo la seguente relazione:

$$\tau + (\beta * B),$$

dove $\beta$ e $\tau$ sono gli id locali associati al thread block e al thread, con $\beta < B$ e $\tau < T$, mentre $B$ e $T$ rappresentano rispettivamente il numero di thread block e il numero di thread per blocco che devono essere utilizzati.
Nel Listato 2.2, scritto in linguaggio di programmazione CUDA C, è riportato un esempio di calcolo del thread id globale utilizzando gli id locali del thread e del blocco, dove *threadIdx.x*, *blockIdx.x* e *blockDim.x* corrispondono rispettivamente ai valori di $\tau$, $\beta$ e $B$.

### Gerarchie di memoria

Attraverso l'astrazione software CUDA mette a disposizione tre diverse tipologie di memorie [9]:

- registri: rappresentano la memoria privata di ogni singolo thread, sono estremamente veloci dato che risiedono all'interno del chip. Vengono utilizzati per memorizzare le variabili locali dichiarate da ogni thread;
- memoria condivisa: la velocità di accesso è pressoché simile a quella dei registri poiché anch'essa è presente all'interno del chip e viene suddivisa fra i thread block schedulati sullo SM, rendendo possibile lo scambio di informazioni intra-blocco evitando in alcuni casi l'accesso alla memoria globale quando si tratta di utilizzare dati utili solamente all'elaborazione interna al blocco stesso;

- memoria globale: è il tipo di memoria più lenta poiché non è presente direttamente all'interno del chip, ma è un'unità DRAM esterna con grande capacità di memorizzazione. Solitamente è utilizzata per le comunicazioni inter-blocco e per lo scambio di dati fra CPU e GPU.

**Gerarchie di esecuzione**

Come descritto in precedenza, un kernel specifica un numero di thread che andranno ad elaborare i dati, questa operazione si traduce fisicamente nell'esecuzione dei thread appartenenti a un thread block su uno SM. All'interno di uno SM possono risiedere diversi thread block (che non potranno essere migrati su altri SM) che verranno eseguiti in modo concorrente. Questa scelta è dovuta al fatto che, così facendo, è possibile utilizzare la memoria condivisa presente all'interno di ogni SM, infatti sia il file dei registri che la memoria Condivisa vengono suddivisi equamente fra tutti i thread block in esecuzione su uno stesso SM, inoltre è disponibile la sincronizzazione fra thread appartenenti ad un thread block tramite la primitiva *__syncthreads()*.

# Capitolo 3

# Risultati

## 3.1 Definizione del modello

La simulazione del fenomeno di proliferazione cellulare nella LMA prevede la creazione di una popolazione iniziale di cellule basata su un istogramma fornito in input, denominato $H(0)$ e contenente coppie di valori $(\varphi_i, \psi_i)$, con $i < |H(0)|$ e $\varphi_i \in \mathbb{R}, \psi_i \in \mathbb{N}$ indicanti rispettivamente il valore di fluorescenza delle cellule rilevato dalle misurazioni in laboratorio e la frequenza con il quale esso si presenta all'interno dei campioni analizzati.
La popolazione iniziale di cellule, denominata $X_0$, è rappresentabile tramite un array di lunghezza $L$ pari alla seguente formula:

$$L = \sum_{i=0}^{|H(0)|-1} \psi_i$$

La popolazione iniziale è rappresentato in Figura 3.1, dove si può notare che ad ogni elemento appartenente a un sottoinsieme di dimensione $\psi_i$ viene assegnato il corrispondente valore di fluorescenza $\varphi_i$. Il fenomeno di divisione cellulare è modellabile tramite un albero binario bilanciato, come mostrato in Figura 3.2, dove ogni nodo rappresenta una cellula della popolazione con valore di fluorescenza $\varphi_i$ dimezzato rispetto alla cellula dalla quale ha avuto origine e ogni ramo indica l'evento di proliferazione corrispondente. $X_0$ dà origine a $L$ alberi di divisione, le cui radici definiscono un insieme di primo livello di cellule proliferanti; e di conseguenza, ogni livello successivo degli alberi rappresenterà un nuovo stadio di proliferazione cellulare avente popolazione con numerosità pari a $L * 2^i$, con $i$ indicante il livello preso in considerazione. Ne consegue che ad ogni livello $i > 0$ corrisponde un array $X_i$ i cui elementi sono le nuove cellule ottenute dalla divisione cellulare avvenuta a partire da $X_{i-1}$. In definitiva, la totalità dei fenomeni di proliferazione è descritta da un insieme di array del tipo $X_i$ aventi come elementi tutte e sole le cellule corrispondenti ad uno specifico livello $i$ degli alberi di proliferazione generati a partire da $X_0$.

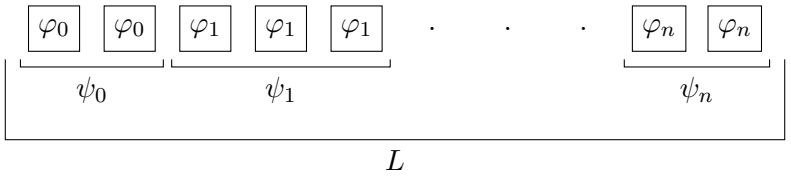



FIGURA 3.1: Rappresentazione della popolazione iniziale di cellule $X_0$, con i rispettivi valori di fluorescenza $\varphi_i$, tramite un array unidimensionale di lunghezza $L$

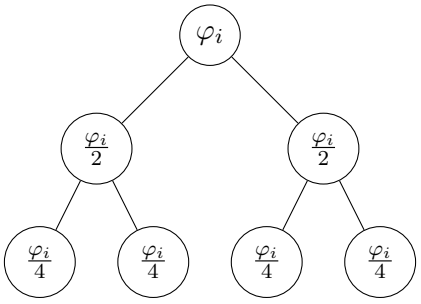



FIGURA 3.2: Fenomeno di divisione cellulare rappresentato tramite albero binario bilanciato dove ogni nodo possiede un valore di fluorescenza $\varphi_i$ dimezzato rispetto al nodo precedente

## 3.2 Algoritmo parallelo

La popolazione di cellule è modellata tramite un array, una delle strutture dati più adatte ad essere parallelizzate su GPU, dato che è possibile demandare la computazione di ogni elemento dell'array ad uno specifico thread. Sia la creazione che la simulazione della proliferazione fanno uso solamente di array unidimensionali, dunque l'obiettivo è l'implementazione di un algoritmo in grado di massimizzare il parallelismo necessario alla computazione dell'insieme di array utile a portare a termine la simulazione.

### 3.2.1 Creazione della popolazione iniziale

L'istogramma $H(0)$ che riporta i valori di fluorescenza con la rispettiva frequenza è salvato su un file di testo che deve essere letto in modo sequenziale. Una volta letto l'intero file, viene creato un array i cui elementi saranno le coppie di valori $(\varphi_i, \psi_i)$. Nota la frequenza $\psi_i$ per ogni valore di fluorescenza $\varphi_i$, è possibile invocare l'esecuzione di un kernel per ogni coppia di valori in modo tale che ogni thread si occupi di generare una cellula avente valore di fluorescenza $\varphi_i$ come rappresentato in Figura 3.3. Oltre al valore di fluorescenza, ogni cellula è caratterizzata da un tipo fra i seguenti [1]: *inattiva, proliferante, proliferazione lenta, proliferazione rapida*, a cui sono associati specifici valori, tranne per la tipologia di cellule inattive, dato che per queste cellule non si verifica mai un evento di divisione:

- intervallo medio di divisione: valore indicante il tempo in ore necessario per il verificarsi di un evento di divisione cellulare;

- deviazione standard dell'intervallo medio di divisione: deviazione standard del valore descritto in precedenza.

Tramite i valori appena descritti, viene creata per ogni nuova cellula una variabile timer $\mu$, derivata da una distribuzione normale, e utilizzata per incrementare il valore di una variabile $\tau$ rappresentante il tempo di vita delle cellule generate da un evento di divisione cellulare.


FIGURA 3.3: Esecuzione di un kernel formato da $\psi_i$ thread necessari alla creazione di esattamente $\psi_i$ cellule aventi $\varphi_i$ come valore di fluorescenza

### 3.2.2 Simulazione della proliferazione cellulare

La popolazione iniziale $X_0$ contiene elementi $X_{0,i}$, ognuno dei quali con l'avanzamento della simulazione dà origine a due nuovi elementi che rappresentano le nuove cellule generate a seguito di un evento di divisione cellulare.
Per la simulazione viene invocato un kernel specifico tale che $X_{0,i}$ sia un elemento dell'array iniziale con $i < L$, a cui è associato un certo valore $\varphi_j$ con $j < |H(0)|$. Denotiamo con $X_1$ un array rappresentante la nuova popolazione cellulare. Dal kernel invocato, per ogni $X_{0,i}$ viene dedicato un GPU thread, che si occuperà di creare due nuove cellule con fluorescenza dimezzata rispetto alla fluorescenza della cellula originaria e genererà il nuovo timer $\mu$ delle cellule sulla base del tipo di cellula considerata.
Le nuove cellule sviluppate dalla proliferazione di $X_{0,i}$ saranno individuate dai seguenti elementi

$$X_{1,(2*i)}$$

$$X_{1,(2*i+1)}$$

Esse avranno il proprio tempo di vita $\tau$ incrementato secondo la seguente formula

$$\tau[X_{1,(2*i)}] = \tau[X_{0,i}] + \mu[X_{0,i}],$$

dove $\mu[X_{0,i}]$ indica il tempo dopo il quale la cellula $X_{0,i}$ procede alla propria divisione.
La simulazione prevede un tempo massimo $\tau_{max}$ e una soglia minima di fluorescenza $\varphi_{min}$ entro cui procedere alla divisione di una cellula, dunque il fenomeno appena descritto avviene solamente se $\tau[X_{0,i}] + \mu[X_{0,i}] \leqslant \tau_{max} \wedge \varphi[X_{0,i}]/2 \geqslant \varphi_{min}$. Questa restrizione implica che all'interno di $X_1$ alcuni elementi non vengano computati. Per ovviare a questo inconveniente è stato deciso di assegnare ad ogni cellula uno stato locale $\sigma \in \{Alive, Inactive, Remove\}$, che indica se la cellula

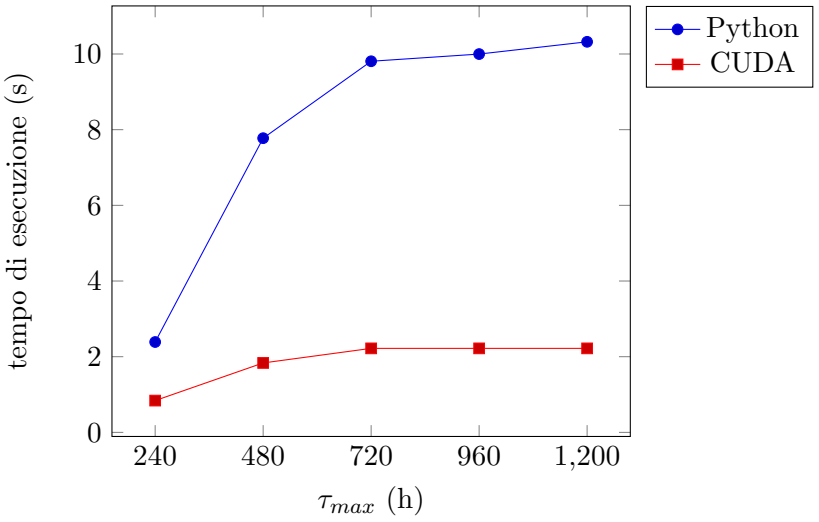



FIGURA 3.4: Confronto dei tempi di esecuzione (in secondi) dell'algoritmo sviluppato in Python e quello sviluppato in CUDA C, utilizzando differenti valori di tempo massimo di proliferazione $\tau_{max}$

può oppure non può più procedere ulteriormente con la divisione.
Sia $\sigma$ l'array degli stati di ogni cellula, allora è possibile riassumere l'assegnazione degli stati come segue

$$\tau[X_{0,i}] + \mu[X_{0,i}] \leqslant \tau_{max} \longrightarrow \sigma[X_{1,(2*i)}] = \sigma[X_{1,(2*i+1)}] = Alive$$

$$\tau[X_{0,i}] + \mu[X_{0,i}] \geqslant \tau_{max} \longrightarrow \sigma[X_{1,(2*i)}] = Inactive \wedge \sigma[X_{1,(2*i+1)}] = Remove$$

$$\varphi[X_{0,i}]/2 \leqslant \varphi_{min} \longrightarrow \sigma[X_{1,(2*i)}] = \sigma[X_{1,(2*i+1)}] = Remove$$

Una volta che il kernel ha terminato l'esecuzione di tutti i thread, il controllo passa nuovamente alla CPU. L'implementazione del metodo a stati risulta essere necessario poiché, a seguito dell'evento di proliferazione cellulare, si deve procedere all'aggiornamento dei risultati rimuovendo da $X_1$ le cellule *Inactive* e aggiungendole ad un nuovo array dei risultati denominato $P$. Si procede inoltre all'eliminazione delle cellule con stato *Remove*, in quanto esse possiedono un valore di fluorescenza $\varphi$ inferiore alla soglia $\varphi_{min}$. L'array risultante conterrà solamente cellule *Alive*, dunque è possibile utilizzare $X_1$ come nuovo array iniziale e reiterare il processo appena descritto fino a quando non saranno più presenti cellule *Alive*. Terminate le iterazioni l'array $P$ conterrà tutte e sole le cellule che hanno superato il limite temporale $\tau_{max}$, dunque è possibile calcolare l'istogramma dei risultati utilizzando le funzioni definite dalle librerie Thrust [2] di CUDA e salvare il risultato ottenuto su file.
Come è possibile notare in Figura 3.4, questa implementazione permette di ottenere un cospicuo aumento delle performance rispetto alla versione non parallela dell'algoritmo, ma non è ancora sufficiente per quanto riguarda i risultati che si vogliono ottenere e per giustificare la scelta di utilizzo di questa versione del simulatore a scapito della versione implementata per CPU.

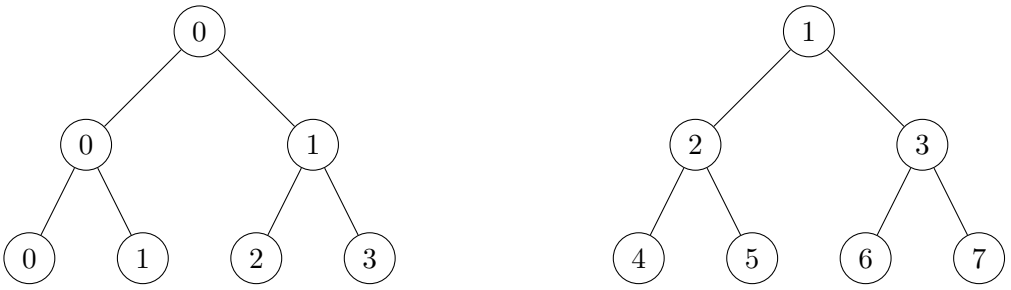



FIGURA 3.5: Rappresentazione di due alberi di proliferazione, le cui radici rappresentano le cellule appartenenti alla popolazione iniziale $X_0$ mentre ogni livello comprende nodi raffiguranti cellule appartenenti a nuove popolazioni cellulari

### 3.2.3 Ottimizzazione dell'implementazione

Sebbene la soluzione descritta in precedenza sia corretta, è tuttavia evidente la necessità di cercare di migliorare ulteriormente l'algoritmo in modo da renderlo ancora più performante, in quanto uno speedup di 8x non è sufficiente a giustificare la scelta di questa soluzione rispetto alla versione procedurale dell'algoritmo. È da tenere in considerazione anche il fatto che l'algoritmo implementato in CUDA fa ancora un utilizzo intensivo della CPU per quanto riguarda l'aggiornamento dei risultati tramite iterazione dell'array della popolazione $X_n$. Detto questo è necessario implementare alcune ottimizzazioni per incrementare le perfomance e rendere questa soluzione preferibile rispetto all'implementazione Python.

**Parallelismo dinamico**

NVIDIA dalla versione *5.0* di CUDA e su architetture hardware a partire dalla versione *compute_35* ha introdotto la possibilità di invocare l'esecuzione di un kernel direttamente dal codice destinato ad un altro kernel. Questa pratica è definita *Parallelismo Dinamico* [8] in quanto permette di computare strutture intrinsecamente ricorsive, come nel caso del *Radix sort* [10].

L'idea è utilizzare il parallelismo dinamico per computare non solo il primo livello degli alberi di proliferazione cellulare, ma proseguire attraverso i livelli successivi fino al termine della simulazione. Per effettuare questa operazione è necessario sfruttare al massimo il concetto di thread block dato che CUDA offre la possibilità di sincronizzare tutti i thread appartenenti ad un blocco tramite la primitiva *__syncthreads()* [8]. L'array $X_1$ viene computato da un numero $\lceil L/2^{10} \rceil$ di thread block, in seguito invece di terminare l'iterazione e aggiornare l'array dei risultati $P$, è possibile calcolare gli array successivi $X_2 ... X_n$ suddividendoli a loro volta in thread block, ovvero sincronizzare i thread tramite la primitiva *__syncthreads()* e successivamente invocare l'esecuzione di un nuovo kernel utilizzando però 2 thread block invece che 1 dato che dal thread block sincronizzato che si sta prendendo in considerazione sono state generate $2 * 2^{10}$ cellule appartenenti alla nuova popolazione.

In Figura 3.5 sono raffigurati due alberi di proliferazione, dove ogni nodo rappresenta l'indice di una cellula all'interno dell'array della popolazione $X_i$ corrispondente al livello $i$ degli alberi considerato, generando le seguenti popolazioni cellulari:

- $X_0 = [0, 1]$
- $X_1 = [0, 1, 2, 3]$
- $X_2 = [0, 1, 2, 3, 4, 5, 6, 7]$

Si denoti con $B_{i,j}$ un generico thread block dove $i$ rappresenta la popolazione $X_i$, mentre $j$ l'indice del blocco. Quindi $B_{0,0} = \{0, 1\}$ sarà il blocco che date le cellule $\{0, 1\} \subseteq X_0$ si occupa di computare le figlie risultanti dalla divisione, ovvero $\{0, 1, 2, 3\} \subseteq X_1$. Al termine della computazione di $B_{0,0}$, viene invocata la primitiva *__syncthreads()* ed eseguito un nuovo kernel, che istanzierà i blocchi $B_{1,0} = \{0, 1\}$ e $B_{1,1} = \{2, 3\}$, che a loro volta eseguiranno $B_{2,0} = \{0, 1\}$, $B_{2,1} = \{2, 3\}$, $B_{2,2} = \{4, 5\}$ e $B_{2,3} = \{6, 7\}$.
Questo metodo accelera di molto l'esecuzione dell'algoritmo, ma introduce un ulteriore problema: non è possibile conoscere ad inizio simulazione la profondità esatta degli alberi di proliferazione necessaria a garantire lo svolgimento di tutti i fenomeni di divisione cellulare, dato che si tratta di una simulazione di tipo stocastico.
Un altro problema che è sorto è la limitazione hardware del parallelismo dinamico, dettata dal fatto che è possibile utilizzare solamente un massimo di 24 livelli di ricorsione. Questo limite ci costringe a dover eventualmente terminare la computazione di alcune popolazioni cellulari qualora rappresentino un livello di proliferazione superiore alla limitazione hardware, iterando nuovamente come se $X_n$ rappresentasse una nuova popolazione iniziale $X_0$ fino a raggiungere il termine della simulazione.

Dal grafico in Figura 3.6 è possibile vedere l'aumento significativo delle performance rispetto alla versione che non presenta l'utilizzo del parallelismo dinamico. Questo è dovuto al fatto che la sincronizzazione di tutti i blocchi della GPU effettuata al termine di ogni iterazione e il trasferimento dei dati da GPU a CPU per l'aggiornamento dell'array dei risultati sono particolarmente onerosi.

**Bounding**

Sebbene il parallelismo dinamico introduca un enorme vantaggio, porta con sè una limitazione importante, ovvero il fatto che un kernel non termina la sua esecuzione fino a quando tutti i kernel invocati da quest'ultimo non sono anch'essi terminati. Durante la computazione degli alberi questo può portare ad alcuni rallentamenti, poiché è inutile computare rami le cui cellule sono state marcate come *Inactive* o *Remove* dato che non contribuiscono alla creazione del nuovo livello di proliferazione. Utilizzando la Shared Memory[12] , ovvero una zona di memoria

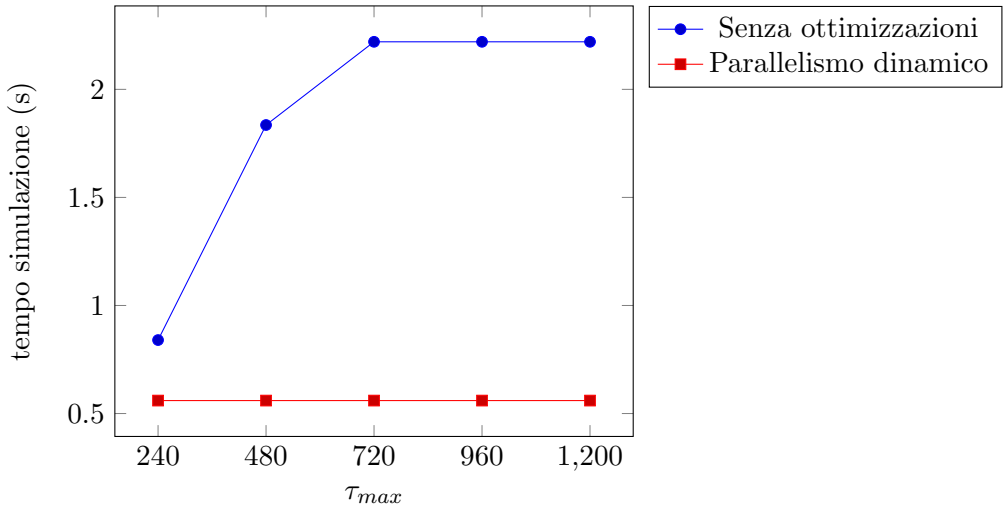
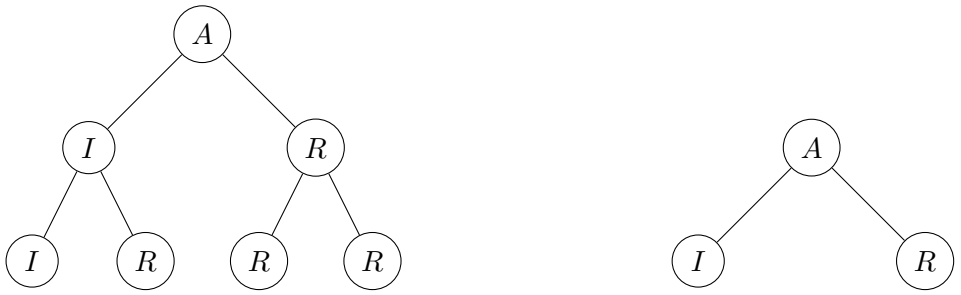



FIGURA 3.6: Confronto dei tempi di esecuzione (in secondi) dell'algoritmo parallelo fra la prima versione senza ottimizzazioni e la versione che implementa il parallelismo dinamico, utilizzando differenti valori di tempo massimo di proliferazione $\tau_{max}$




(A) Senza bounding (B) Bounding

FIGURA 3.7: Confronto di alberi di proliferazione cellulare utilizzando il metodo del bounding

condivisa tra tutti e soli i thread appartenenti ad uno stesso thread block, è possibile settare un flag indicante l'avvenuto evento di proliferazione cellulare. Se ci si trovasse nel caso in cui nessun evento si è verificato, sarebbe inutile invocare un nuovo kernel, di conseguenza la computazione dei successivi sotto-alberi viene interrotta per consentire la preventiva terminazione dei kernel precedenti.

Siano $A = Alive$, $I = Inactive$, $R = Remove$, allora il metodo implementato produce i risultati rappresentati in Figura 3.7. Come è possibile notare, l'albero risultante dall'applicazione del bounding presenta un numero inferiore nodi, ciò equivale al fatto di risparmiare risorse evitando la computazione di sotto-alberi inutili ai fini della simulazione. L'aumento delle performance ottenuto tramite il bounding non è significativo, ma questo tipo di ottimizzazione migliora la gestione delle risorse computazionali relative alla GPU, in particolare all'utilizzo della memoria dopo l'implementazione del parallelismo dinamico. Infatti quando un kernel padre invoca un kernel figlio, il padre dovrà attendere la fine di tutti i suoi kernel figli per poter procedere con l'esecuzione del codice e terminare a sua volta. Questa tecnica di bounding risulterà molto più utile in seguito, con l'implementazione dell'ultima tecnica di ottimizzazione del simulatore.

### Inferenza del livello di profondità

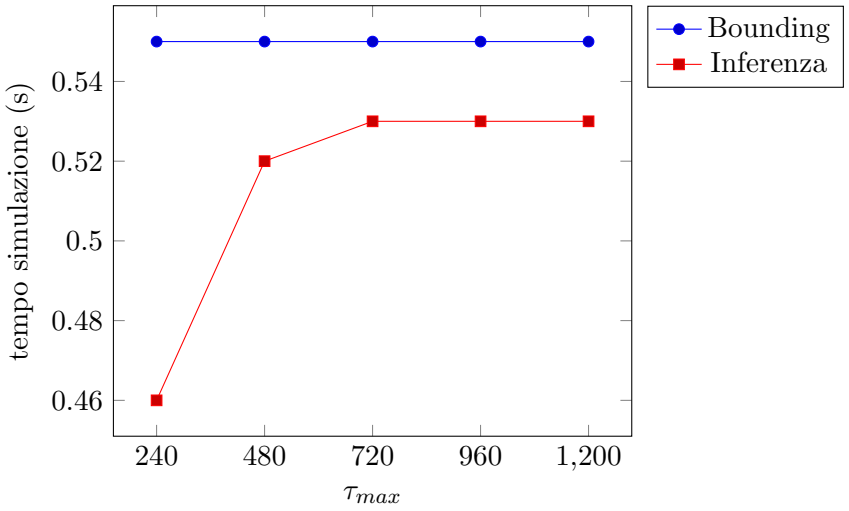



FIGURA 3.8: Confronto dei tempi di esecuzione (in secondi) dell'algoritmo parallelo fra la versione che implementa le ottimizzazioni descritte fino al metodo del bounding e la versione che oltre a quest'ultimo utilizza anche l'inferenza del numero di livelli necessari alla simulazione, utilizzando differenti valori di tempo massimo di proliferazione $\tau_{max}$

Sebbene CUDA ci ponga un livello massimo di nesting pari a 24 livelli, nei modelli presi in considerazione per queste simulazioni, difficilmente viene superata la soglia dei 5 eventi di proliferazione per ogni cellula, ma la versione dell'algoritmo dopo l'implementazione del parallelismo dinamico prevede l'allocazione di $N$ livelli di proliferazione pari al numero massimo di cellule che è possibile mantenere sulla memoria dedicata alla GPU. Dunque se è presente sufficiente spazio per allocare 10 livelli di profondità essi vengono allocati, consumando più risorse del dovuto. Non è nemmeno possibile utilizzare la memoria di *swap* dato che le GPU non supportano questa funzionalità. È necessario quindi trovare un modo per inferire nel modo più accurato possibile il numero di livelli necessari alla simulazione. Un modo banale potrebbe essere quello di dare la possibilità all'utente di specificare il numero di livelli da utilizzare durante la simulazione, a patto che sia al corrente del numero medio di divisioni necessarie per ogni cellula appartenente ad $X_0$ al fine di portare a termine la simulazione. Un altro metodo risulta essere la possibilità di inferire il numero dei livelli necessari utilizzando i parametri forniti in ingresso alla simulazione. Per poter calcolare i diversi tipi di cellule, si utilizza un array di parametri contenente le distribuzioni uniformi $\pi$ dei vari tipi di cellule che è possibile avere all'interno della popolazione iniziale. Ogni tipo di cellula possiede un tempo medio $\mu$ dopo il quale avviene la divisione, il quale segue una distribuzione normale. È possibile pensare di utilizzare il tempo medio $\mu_i$ del tipo di cellula $\tau_i$ che ha probabilità $\pi_i$ maggiore per calcolare un numero approssimativo di livello di divisioni tali per cui venga superato il $\tau_{max}$ definito per la simulazione.

Sia $\mu_{max}$ il tempo medio di divisione del tipo di cellula avente $max(\pi)$. Allora è possibile stimare

che il numero $\eta$ dei livelli di profondità dell'albero è dato dalla seguente relazione

$$\eta : \sum_{i=1}^{\eta-1} \mu_{max} \leqslant \tau_{max} \leqslant \sum_{i=1}^{\eta} \mu_{max}$$

Questo valore è utile per l'ottimizzazione del numero di livelli necessario per portare a termine la simulazione mediante una sola iterazione. Essendo però un processo di tipo stocastico, è possibile che il valore calcolato $\eta$ non rispecchi a pieno la realtà che si vuole simulare.
Nel grafico in Figura 3.8 si può notare un incremento di prestazioni fino ad un valore di $\tau_{max}$ di 720, questo perché sebbene venga applicata l'inferenza dei parametri, l'algoritmo prevede l'istanziamento di $\eta$ livelli solamente se è disponibile abbastanza memoria all'interno della GPU. Infatti con una valore di $\tau_{max} > 720$ non è presente nessun aumento di prestazioni poiché la GPU utilizzata non ha abbastanza memoria per istanziare un numero di livelli pari a $\eta$ inferito dai parametri iniziali. Quindi risulta sicuramente un vantaggio l'utilizzo di GPU con molta memoria dedicata a disposizione.

### Calcolo dell'istogramma finale

Sebbene le performance siano aumentate, il software fa ancora molto affidamento sulla CPU per il calcolo dell'istogramma finale. Ad ogni iterazione l'array della popolazione $X_n$ viene trasferito attraverso la CPU in memoria centrale per essere filtrato e per aggiornare i risultati dell'istogramma. Questo processo è possibile grazie alla funzione *cudaMemcpy()* che si occupa del trasferimento dei dati tra CPU e GPU. Diverse chiamate a questa funzione generano un lavoro intensivo che deve essere effettuato, quindi impatta ulteriormente sullo speedup della simulazione. L'obiettivo è quindi cercare di minimizzare il numero di trasferimenti tramite *cudaMemcpy()*.
La soluzione risiede nella possibilità di calcolare l'istogramma direttamente sulla GPU. Assumendo di conoscere un array $\Omega$ ordinato in modo crescente contenente tutti possibili valori di fluorescenza $\varphi$ che una cellula potrà assumere in futuro, è possibile far eseguire ad un thread del kernel (se computando una cellula *Active* essa non è più in grado di proliferare) una ricerca binaria su $\Omega$ e aggiornare il valore di frequenza $\psi$ corrispondente alla fluorescenza trovata, tramite la funzione *atomicAdd*() di CUDA, la quale anche se prevede una sincronizzazione sulla risorsa a cui si vuole accedere, è necessaria dato che ad ora non esiste un altro modo per il calcolo di un istogramma attraverso un algoritmo parallelo.
Conoscendo l'istogramma iniziale $H(0)$ si hanno già a disposizione tutti i valori di fluorescenza che una cellula potrà assumere, infatti $\Omega$ conterrà tutti i valori di $H(0)$ tali che $\varphi_i \geqslant \varphi_{min}$ e i successivi valori $\varphi_i^{-2j}$ con $j \in \mathbb{N}, j > 0$.
Questa tecnica evita il trasferimento dati al termine di ogni iterazione, poiché la frequenza riguardante i valori di fluorescenza delle cellule $Inactive$ è già stata aggiornata durante l'esecuzione

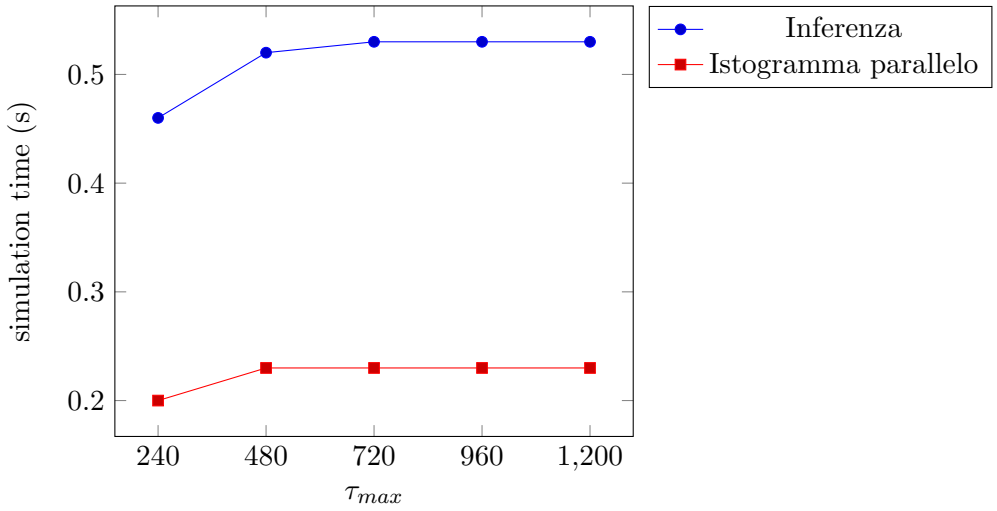



FIGURA 3.9: Confronto dei tempi di esecuzione (in secondi) dell'algoritmo parallelo fra la versione che implementa le ottimizzazioni descritte fino al metodo dell'inferenza dei parametri e la versione che oltre a quest'ultimo implementa anche il calcolo parallelo dell'istogramma finale, utilizzando differenti valori di tempo massimo di proliferazione $\tau_{max}$

del kernel. Mediante questa tecnica la $cudaMemcpy()$ viene utilizzata solamente a inizio e fine simulazione per il salvataggio dei risultati.

Come si evince dal grafico in Figura 3.9, questa è la tecnica di ottimizzazione che incrementa maggiormente le performance dell'algoritmo, dato che risolve quasi totalmente il problema del collo di bottiglia causato dal trasferimento bidirezionale delle informazioni per l'avanzamento della simulazione.

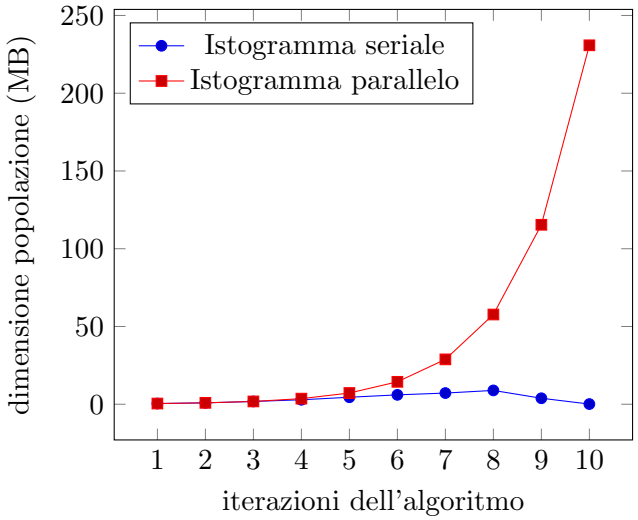


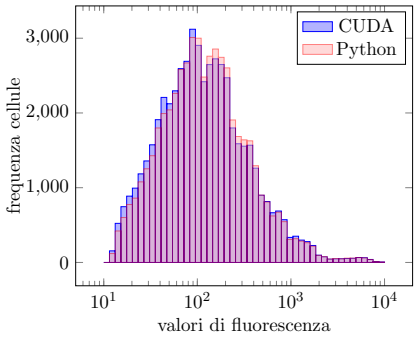

FIGURA 3.10: Confronto dell'utilizzo di memoria fra la versione dell'algoritmo che prevede il calcolo seriale dell'istogramma con rimozione delle cellule non *Alive* e il calcolo parallelo dell'istogramma dei risultati senza filtraggio della popolazione cellulare al termine di $n$ iterazioni dell'algoritmo

### 3.2.4 Gestione della memoria

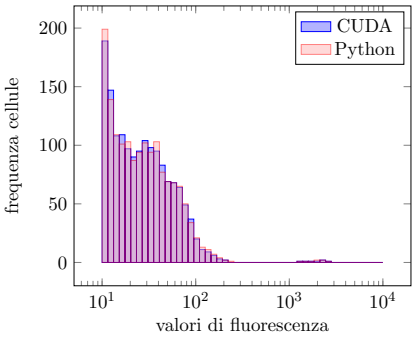
L'implementazione delle ottimizzazioni descritte in precedenza ha migliorato di gran lunga le performance dell'algoritmo, ma ha introdotto un problema, ovvero la crescita esponenziale delle risorse da allocare. Infatti data la numerosità $L$ della popolazione iniziale $X_0$, essa cresce esponenzialmente con il numero dei livelli dell'albero $L * 2^i$ con $i \in \mathbb{N}, i > 0$. Inizialmente questa crescita era mitigata dal fatto che dopo ogni iterazione la popolazione veniva filtrata, quindi procedendo verso il termine della simulazione, l'utilizzo di memoria veniva bilanciato dalla poca numerosità delle cellule *Alive* rimanenti. Dopo l'introduzione dell'ultima ottimizzazione, sebbene le cellule *Inactive* e *Remove* non vengano prese in considerazione per la computazione, sono comunque presenti ancora all'interno delle diverse popolazioni, dato che dopo ogni iterazione la memoria non viene più filtrata in quanto non più trasferita sulla CPU.
La crescita esponenziale dell'occupazione di memoria visibile in Figura 3.10 potrebbe essere un problema nel momento in cui il tempo $\tau_{max}$ inizia a crescere ulteriormente. La soluzione immediata a questo problema potrebbe essere di natura economica, ovvero l'acquisto di GPU moderne con molta memoria a disposizione. Il problema però persiste, quindi una possibile soluzione potrebbe essere quella di computare ad un certo punto solamente un sottoinsieme della popolazione e trasferire i restanti elementi sulla CPU, per poi elaborarli nuovamente quando il primo sottoinsieme di popolazione è stato elaborato. Questo però introdurrebbe nuovamente la necessità di utilizzare in modo assiduo $cudaMemcpy()$, però fino ad ora pare essere l'unica soluzione plausibile.

## 3.3 Confronto con dati sperimentali

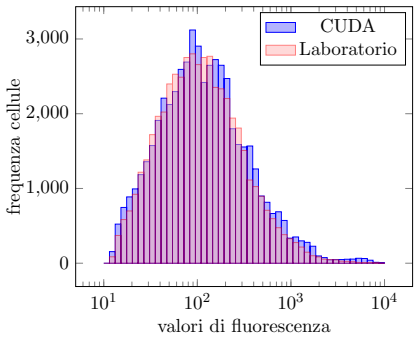
Lo scopo della simulazione è verificare la correttezza del modello utilizzato sulla base di dati reali provenienti da esperimenti effettuati in vivo in laboratorio.
In Figura 3.11 è possibile vedere la comparazione tramite overlap dei risultati ottenuti dalle simulazioni fra l'algoritmo di riferimento sviluppato in Python e la versione implementata in CUDA. Nei grafici sono presenti aree in cui i valori di frequenza si discostano tra loro di alcune unità. Questa variazione è dovuta al fatto che la simulazione deve elaborare eventi di tipo stocastico, dunque il numero di cellule aventi un determinato tipo e il timer di divisione non saranno mai esattamente uguali per ogni simulazione che si andrà ad effettuare, e questa situazione è appunto evidenziata dalle aree del grafico dove è assente l'overlap dei valori.



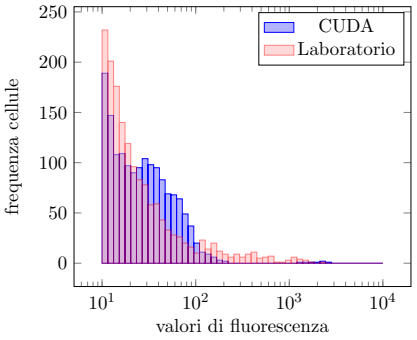

(A) Confronto simulazioni Python e CUDA con $\varphi_{min} = 11.0, \tau_{max} = 240$




(B) Confronto simulazioni Python e CUDA con $\varphi_{min} = 8.7, \tau_{max} = 504$




(C) Confronto simulazione CUDA con $\varphi_{min} = 11.0, \tau_{max} = 240$ e dati sperimentali




(D) Confronto simulazione CUDA con $\varphi_{min} = 8.7, \tau_{max} = 504$ e dati sperimentali

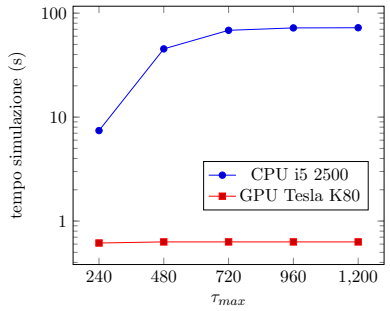
FIGURA 3.11: Confronto dei valori di fluorescenza sperimentali ricavati in laboratorio e quelli ottenuti dalle simulazioni utilizzando l'algoritmo sviluppato in Python e in CUDA con diversi parametri iniziali

## 3.4 Performance

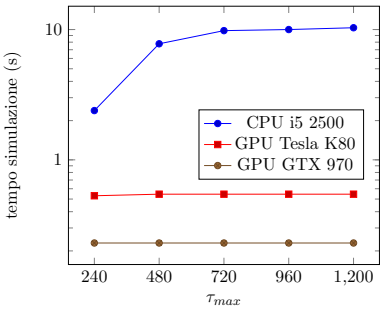
L'implementazione di un modello ad albero per la parallelizzazione di eventi di proliferazione cellulare si è rivelata una tecnica efficiente per questo tipo di simulazione, in Figura 3.12

sono presenti i risultati ottenuti per quanto riguarda i tempi necessari alla computazione della simulazione modificando il parametro $\tau_{max}$.

Come è possibile notare dal grafico in Figura 3.13 la scheda GTX 970 è più performante della Tesla K80. Questo è dato dal fatto che la GTX 970 possiede un clock di 1050 MHz, mentre la Tesla K80 solamente di 562 MHz, dunque frequenza quasi dimezzata. Questo impatta leggermente sulle performance a causa della ricerca binaria implementata nel calcolo dell'istogramma finale, dato che un thread deve eseguire un loop per la ricerca della fluorescenza $\varphi_i$ all'interno dell'array delle frequenze $\Omega$. Sebbene la ricerca binaria sia nell'ordine di $O(\log_2 n)$, con un clock minore si rischia di avere un leggero calo di performance rispetto a schede con clock maggiori.

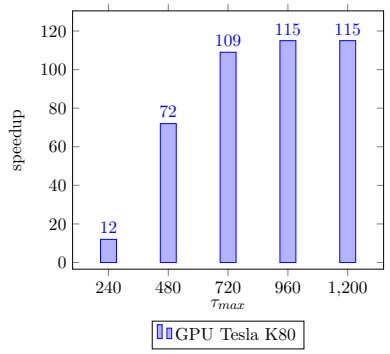



(A) Simulazione effettuata a partire da $H(0)$ con valore minimo di fluorescenza rilevato $\varphi_{min} = 11.0$

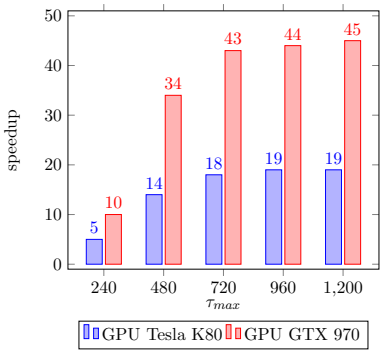



(B) Simulazione effettuata a partire da $H(0)$ con valore minimo di fluorescenza rilevato $\varphi_{min} = 8.7$

FIGURA 3.12: Confronto dei tempi (in secondi) necessari per il termine della simulazione fra l'algoritmo sviluppato in Python e in CUDA utilizzando GPU diverse con diverso parametro iniziale $\varphi_{min}$ per la simulazione, aumentando il tempo massimo $\tau_{max}$ per la proliferazione cellulare




(A) Speedup della simulazione con valore iniziale $\varphi_{min} = 11.0$ e differenti valori di $\tau_{max}$




(B) Speedup della simulazione con valore iniziale $\varphi_{min} = 8.7$ e differenti valori di $\tau_{max}$

FIGURA 3.13: Confronto dello speedup ottenuto tramite l'algoritmo implementato in CUDA rispetto alla versione non parallela in Python ed eseguito su GPU con caratteristiche hardware differenti fra loro

# Capitolo 4

# Conclusioni

I risultati ottenuti dalle simulazioni effettuate mediante l'utilizzo dell'algoritmo implementato in CUDA messi a confronto con i risultati del medesimo algoritmo codificato in Python hanno confermato la correttezza dell'algoritmo CUDA, dato che utilizza una struttura dati ad albero per l'elaborazione della simulazione, differentemente dalla versione Python che fa uso di una struttura a stack, più intuitiva da implementare e gestire. La scelta di questo tipo di strutture ad albero ha portato un enorme vantaggio in termini prestazionali, poiché è stato scelto di implementare una soluzione che prevedesse l'utilizzo del parallelismo dinamico per verificare la validità di questa metodologia nella simulazione di fenomeni intrinsecamente ricorsivi. Il parallelismo dinamico con molti livelli di profondità risulta quindi un'ottima soluzione quando si tratta di elaborare strutture dati predisposte alla ricorsione, come in questo caso. Sebbene CUDA preveda un limite fisico al numero di livelli di ricorsione disponibili, è degna di nota l'accelerazione fornita alla simulazione. Gli sviluppi futuri prevedono, a fronte della crescita esponenziale della popolazione cellulare, l'ottimizzazione dell'utilizzo della memoria tramite l'implementazione nel software di tecniche per la gestione di un numero di GPU superiore ad uno, in quanto sarebbe possibile demandare la computazione di sottopopolazioni cellulari ognuna ad una GPU dedicata, incrementando ulteriormente le performance del simulatore. Inoltre sarebbe interessante analizzarne lo speedup a fronte di una popolazione iniziale $X_0$ con numerosità estremamente maggiore di quella utilizzata per le simulazioni riportate nella Sezione 3.4, approssimativamente di $5 * 10^4$ elementi.

# Bibliografia


[1] Marco S. Nobile et al. “Modeling cell proliferation in human acute myeloid leukemia xenografts”. manuscript in preparation.

[2] Nathan Bell e Jared Hoberock. “Thrust: A productivity-oriented library for CUDA”. In: *GPU computing gems Jade edition*. Elsevier, 2011, pp. 359–371.

[3] J. M. et al Bernitz. “Hematopoietic stem cells count and remember self-renewal divisions”. In: *Cell* 167(5) (2016), pp. 1296–1309.

[4] Ralph Duncan. “A survey of parallel computer architectures”. In: *Computer* 23.2 (1990), pp. 5–16.

[5] H. et al. Döhner. “Acute myeloid leukemia”. In: *N. Engl. J. Med.* 373(12) (2015), pp. 1136–1152.

[6] Ian Foster et al. “Grid services for distributed system integration”. In: *Computer* 6 (2002), pp. 37–46.

[7] John C Hunter e John A Wertz. *Multi-node cluster computer system incorporating an external coherency unit at each node to insure integrity of information stored in a shared, distributed memory*. US Patent 5,394,555. 1995.

[8] Stephen Jones. “Introduction to dynamic parallelism”. In: *GPU Technology Conference Presentation S*. Vol. 338. 2012, p. 2012.

[9] David Kirk et al. “NVIDIA CUDA software and GPU parallel computing architecture”. In: *ISMM*. Vol. 7. 2007, pp. 103–104.

[10] Duane Merrill e Andrew Grimshaw. “High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing”. In: *Parallel Processing Letters* 21.02 (2011), pp. 245–272.

[11] John Nickolls e William J Dally. “The GPU computing era”. In: *IEEE micro* 30.2 (2010).

[12] Jason Sanders e Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.