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 Unit di modelli di proliferazione cellulare

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

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

**Relazione della prova finale di:**

*Eric Nisoli*

*Matricola 807147*

**Anno Accademico 2017-2018**

# Indice

# Capitolo 1

# Introduzione

# Capitolo 2

# Metodi

## 2.1 Architettura CUDA

CUDA è l'architettura di elaborazione in parallelo progettata e sviluppata da NVIDIA che sfrutta la potenza di calcolo delle GPU (Graphics Processing Units) per aumentare le prestazioni nell'ambito del software computing. L'elaborazione sta lentamente migrando verso il paradigma di *co-processing* su CPU e GPU 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 architettura 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 fenomi 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 (SMs) [4, p. 62]

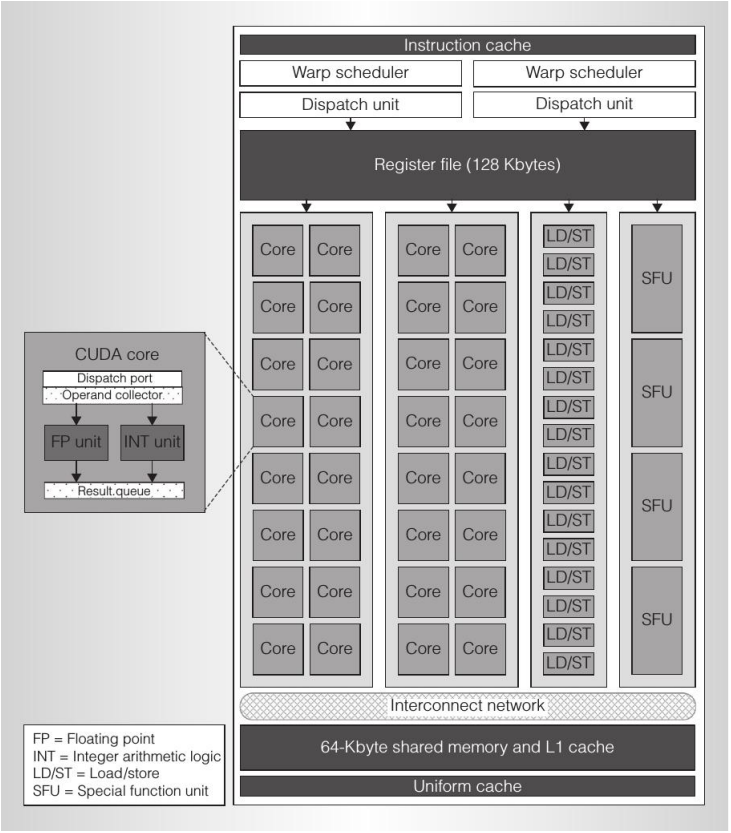



FIGURA 2.1: Streaming Multiprocessor dell'architettura Fermi [4, p. 63]

**Astrazione Software** Dato che la struttura fisica delle schede è in continuo mutamento, 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 molte schede che supportano CUDA.
Il modello astratto definisce tre tipologie di oggetti:

- **Thread**: singole unità di calcolo, eseguono il codice sorgente;
- **Thread block**: insieme logico di thread. I thread appartenenti allo stesso blocco hanno accesso ad un'area di memoria condivisa e accessibile solamente da loro (oltre alla memoria globale della GPU); inoltre è possibile ottenere un livello di sincronizzazione fra i thread del blocco;
- **Grid**: insieme logico di *Thread block*. Non è stata prevista un'area di memoria condivisa da tutta la griglia e fino ad ora non esiste una primitiva per sincronizzare i blocchi di una specifica griglia, è quindi necessario procedere alla sincronizzazione di tutta la GPU.

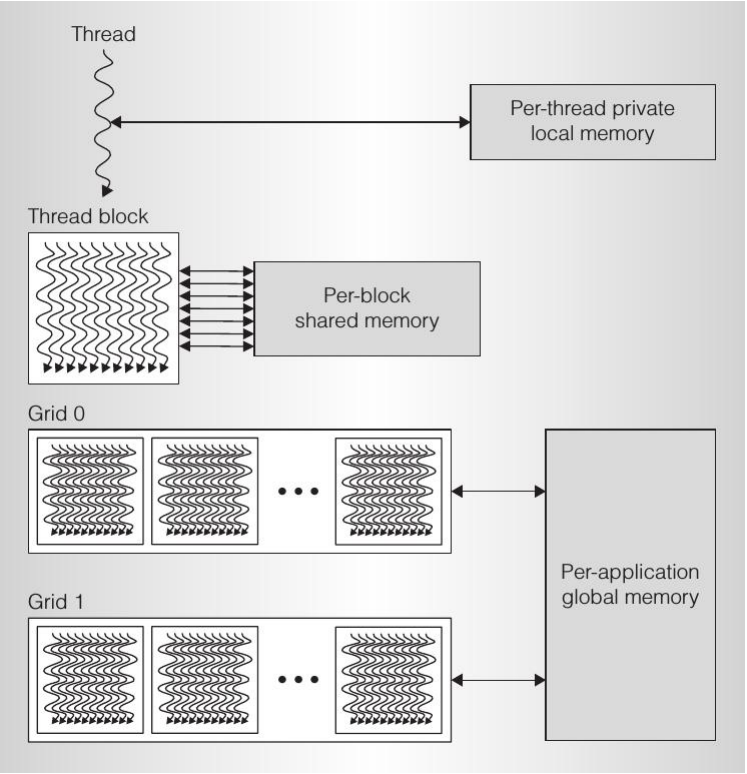



FIGURA 2.2: Schema della gerarchia di *Thread*, *Thread block*, e *Grid* [4, p. 59]

Le procedure che vengono eseguite sulla GPU vengono chiamate *Kernel* (identificabili grazie al prefisso *__global__*) ed è possibile specificarne la dimensionalità, ovvero decidere quanti *Thread*, *Thread block* e *Grid* verranno assegnati all'esecuzione del codice invocato.

```
1  __global__
2  void
3  my_kernel()
4  {
5      // GPU code...
6  }
7
8  // CPU code
9  int main()
10 {
11     int n_blocks = 8; // Example blocks number
12     int n_threads_per_block = 32; // Example threads per block number
13
14     // Invoke Kernel on GPU
15     my_kernel<<<n_blocks, n_threads_per_block>>>();
16 }
```

LISTING 2.1: Esempio di invocazione GPU kernel

Come è possibile notare nell'esempio di codice 2.1, stiamo invocando l'esecuzione di un kernel specificando l'utilizzo di 8 *Thread Block* e 32 *Thread* per blocco (se non viene specificato il numero delle *Grid* il valore di default è 1). In generale il numero totale di *Thread* che verranno utilizzati per la computazione del kernel è

$$\Gamma * B * T$$

dove $\Gamma, B, T$ sono rispettivamente il numero di *Grid*, il numero di *Thread Block* e il numero di *Thread* che vogliamo utilizzare.

La possibilità di decidere la suddivisione del kernel tra griglie e blocchi ci ritorna molto utile nell'elaborazione di strutture dati non particolarmente complesse come possono essere gli array. Infatti è sufficiente invocare un kernel con numero di thread uguale (o maggiore) al numero di elementi dell'array e computare ogni elemento in un thread diverso.
Esiste però un limite al numero di *Thread* per blocco che è possibile dichiarare (nelle nuove versioni di CUDA è *1024*), dunque per ottenere l'*id* globale del thread relativo al kernel eseguito dobbiamo avvalerci anche del numero di *Grid* e *Thread Block* richiesti, secondo la seguente relazione:

$$\tau + (\beta * B) * (\gamma * G)$$

dove $\Gamma, B, T$ sono rispettivamente il numero di *Grid*, il numero di *Thread Block*, e il numero di *Thread* che vogliamo utilizzare e $\gamma, \beta, \tau$ sobo gli *id* locali associati alle *Grid*, *Thread Block* e *Thread* con $\gamma < \Gamma, \beta < B, \tau < T$.

```
1  __global__
2  void
3  my_kernel(int* array)
4  {
5      // Computing current thread global id
6      int id = threadIdx.x + (blockIdx.x * blockDim.x) * (gridIdx.x * gridDim.x);
7
8      // Work with array values
9      array[id] = 0;
10 }
```

Listing 2.2: Esempio di calcolo GPU kernel thread global id

## 2.2 Modellazione del problema

La simulazione del problema prevede la creazione di una popolazione iniziale di cellule basata su un istogramma fornito in input denominato $H(0)$ contenente coppie di valori $(\varphi_i, \psi_i)$, con $\varphi_i \in \mathbb{R}, \psi_i \in \mathbb{N}$ indicanti rispettivamente il valore di fluorescenza rilevato dalle misurazioni in laboratorio e la frequenza con il quale esso si presenta. La popolazione iniziale di cellule è rappresentabile tramite un array di lunghezza pari a

$$L = \sum_{i=1}^{|\varphi|} \psi_i$$

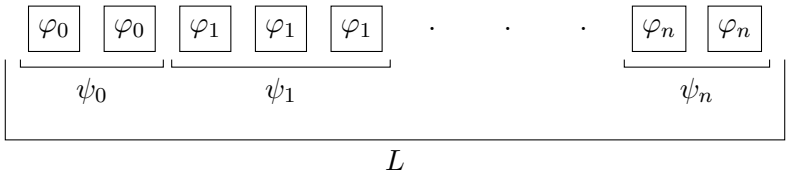



Figura 2.3

Ad ogni fenomeno di divisione cellulare corrisponde la creazione di due nuove cellule aventi valore di fluorescenza dimezzato rispetto alla cellula originaria. Questo comportamento è modellabile tramite un albero binario bilanciato

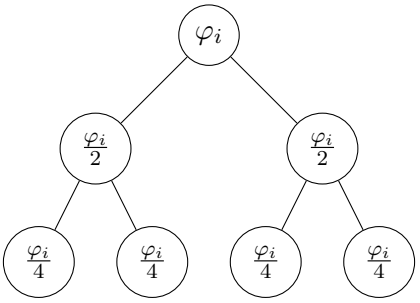



Figura 2.4

La popolazione iniziale di cellule dunque dà vita ad insieme di $L$ alberi di divisione. Il primo livello di questo insieme corrisponde alla popolazione iniziale di cellule, di conseguenza ogni livello successivo rappresenterà un nuovo stadio di proliferazione cellulare avente popolazione con numerosità uguale a $L * 2^i$ con $i$ uguale al livello considerato. Quindi se il primo livello dell'albero corrisponde all'array della popolazione iniziale, allora anche ad ogni livello successivo corrisponde un array i quali elementi sono le nuove cellule ottenute dalla divisione cellulare del livello immediatamente precedente. In definitiva la totalità dei fenomeni di proliferazione è descritta da un insieme di array, dove ognuno racchiude tutte e sole le cellule corrispondenti ad uno specifico livello degli alberi di proliferazione generati a partire dalla popolazione iniziale.

## 2.3 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 il problema rimane solamente la stesura di un algoritmo in grado di massimizzare il parallelismo per computazione dell'insieme di array necessario a portare a termine la simulazione.

### 2.3.1 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 avremo creato un array i cui elementi saranno le coppie di valori $(\varphi_i, \psi_i)$. Ora che è nota la frequenza $\psi_i$ per ogni fluorescenza $\varphi_i$, è possibile invocare l'esecuzione di un *kernel* per ogni coppia di valori tale che al kernel vengano assegnati esattamente $\psi_i$ thread, in modo tale che ognuno di essi si occupi di generare una cellula avente $\varphi_i$ valore di fluorescenza.


Figura 2.5

### 2.3.2 Simulazione della proliferazione cellulare

L'idea generale è che dato l'array della popolazione iniziale $X_0$ sia possibile per ogni elemento $X_{0,i}$ in esso contenuto sviluppare la simulazione creando due nuove cellule figlie partendo da una cellula originaria, ovvero l'elemento selezionato dall'array iniziale.
Per la simulazione viene invocato un *kernel* specifico. Sia $X_{0,i}$ un elemento dell'array iniziale con $i < L$, a cui è associato un certo valore $\varphi_j$ con $j < |\varphi|$ e sia $X_1$ un array rappresentante la nuova popolazione cellulare. Dal *kernel* invocato, per ogni $X_{0,i}$ viene dedicato un GPU thread, che svolgerà le seguenti procedure.
Le nuove cellule generate dalla proliferazione di $X_{0,i}$ saranno

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

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

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

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

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 $\varphi_{min}$ entro il quale procedere alla divisione di una cellula, dunque il fenomeno appena descritto avviene solamente se $\tau[X_{0,i}] + \mu[X_{0,i}] \leqslant \tau_{max}$ oppure se $\varphi[X_{0,i}]/2 \geqslant \varphi_{min}$. 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 $\{Alive, Inactive, Remove\}$ indicatore del fatto che la cellula può oppure non può più procedere ulteriormente con la divisione.
Sia $\sigma$ l'array degli stati di ogni cellula, allora possiamo riassumere l'assegnazione degli stati come segue

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

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

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

Una volta che il kernel ha terminato l'esecuzione di tutti i thread, il controllo viene ritornato alla CPU. Gli stati risultano necessari poichè dopo l'evento di proliferazione cellulare dobbiamo procedere all'aggiornamento dei risultati rimuovendo da $X_1$ le cellule $Inactive$ e aggiungendole ad un nuovo array dei risultati $P$. Procediamo inoltre all'eliminazione delle cellule con stato *Remove* in quanto possiedono un valore di fluorescenza $\varphi$ sotto la soglia $\varphi_{min}$. L'array risultante conterrà solamente cellule *Alive*, dunque possiamo 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 la funzioni definite dalle librerie *Thrust*[1] di CUDA e salvare il risultato ottenuto su file.

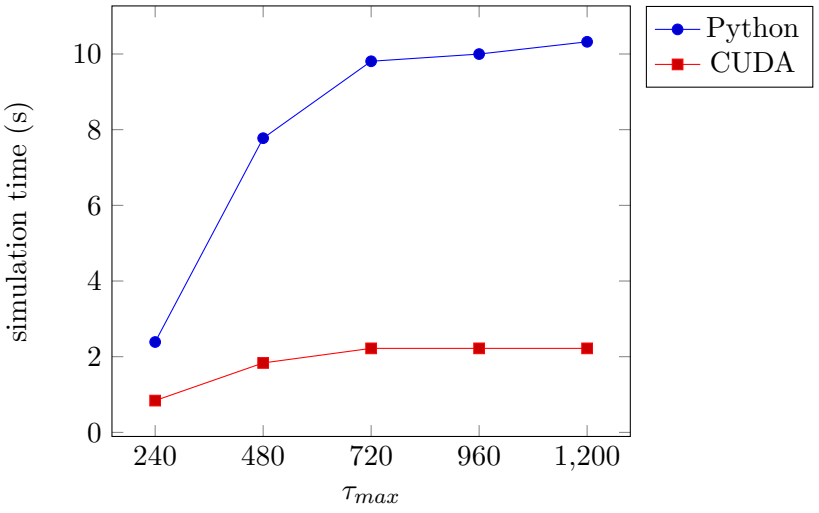



FIGURA 2.6

Come è possibile notare dal grafico, è presente un aumento delle performance rispetto alla versione procedurale dell'algoritmo, ma non è ancora degno di nota per quanto riguarda i risultati che si vogliono ottenere.

### 2.3.3 Acceleranti

Sebbene la soluzione descritta in precedenza sia corretta, come mostrato nei grafici è evidente che bisogna cercare di migliorare ulteriormente l'algoritmo in modo da renderlo ancora più performante, in quanto uno speedup di **8x** non è sufficiente per preferire questa soluzione rispetto ad una procedurale considerando anche il fattore costo per le moderne GPU rispetto alle CPU in commercio. È anche da tenere in considerazione il fatto che l'algoritmo si appoggia ancora molto sulla CPU quando si tratta di iterare l'array delle soluzioni. Detto questo è necessario implementare alcuni *acceleranti* per incrementare le perfomance e rendere questa soluzione preferibile rispetto all'algoritmo scritto per la CPU.

**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 da un altro kernel. Questa pratica è definita *Parallelismo Dinamico*[2] in quanto permette di computare strutture intrinsecamente ricorsive, dato che da un kernel può essere invocato lo stesso kernel con parametri differenti come ad esempio l'implementazione del *radix sort*[3].
L'idea è utilizzare il parallelismo dinamico per computare non solo il primo livello degli alberi di proliferazione cellulare, ma proseguire fino ad arrivare all'ultimo livello, ritornando il controllo alla CPU solo in questo caso. Per effettuare questa operazione bisogna 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()*[2]. L'array $X_1$ viene computato da $\lceil L/2^{10} \rceil$ *Thread Block*, invece di terminare la computazione è possibile continuare a computare gli array successivi $X_2 ... X_n$ a blocchi, ovvero sincronizzare i thread all'interno di un *Thread Block* tramite la primitiva *__syncthreads()* e in seguito invocare l'esecuzione di un nuovo kernel utilizzando però 2 *Thread Block* dato che il blocco corrente ha generato $2 * 2^{10}$ cellule appartenenti al livello successivo dell'albero.

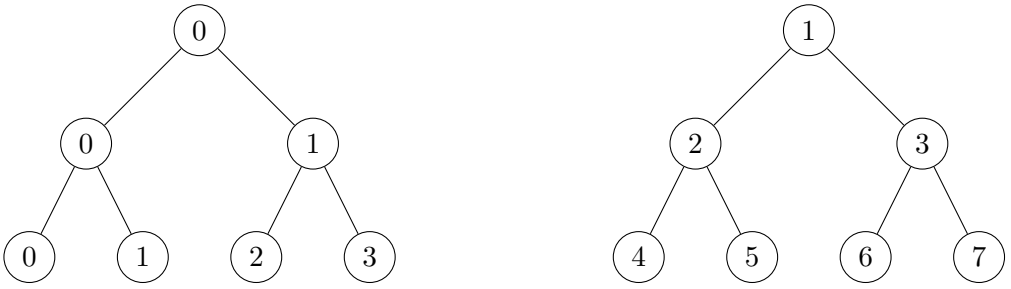



FIGURA 2.7

In figura possiamo vedere due alberi di proliferazione, dove ogni nodo rappresenta l'indice di una cellula all'interno dell'array della popolazione corrispondente al livello degli alberi considerato, come segue:

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

Denotiamo 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: a priori non è possibile conoscere la profondità esatta degli alberi di proliferazione, dato che si tratta di una simulazione di tipo stocastico, dunque l'utilizzo di memoria aumenta esponenzialmente all'aumentare dei livelli di profondità degli alberi.
Un altro problema che è sorto è la limitazione hardware del parallelismo dinamico, ovvero il fatto che è possibile innestare solamente un massimo di **24 livelli** di ricorsione. Questo fatto ci costringe a dover eventualmente terminare la ricorsione e ritornare il controllo alla CPU, iterando nuovamente fino a raggiungere il termine della simulazione.

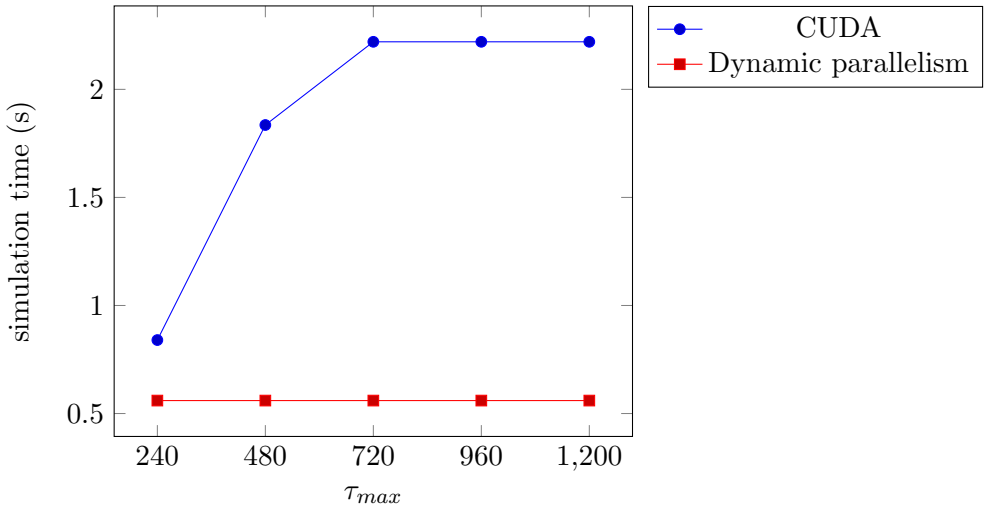
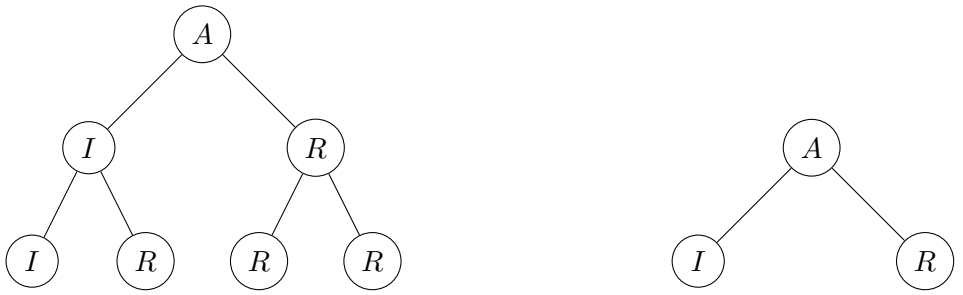



FIGURA 2.8

Dal grafico in figura 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 delle GPU che deve essere effettuata al termine di ogni iterazione è molto onerosa in termini prestazionali, inoltre anche la copia dei dati da GPU a CPU per l'aggiornamento dell'array dei risultati risulta particolarmente time-consuming.

**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, dato che 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. Detto questo risulta utile tenere traccia del fatto che un *Thread Block* abbia oppure no generato nuove cellule. In questo caso ci viene in aiuto la *Shared Memory*[5] , ovvero una zona di memoria condivisa tra tutti i thread appartenenti ad uno stesso *Thread Block*. Utilizzando quest'area di memoria è possibile settare un flag indicante il fatto che è avvenuto un evento di proliferazione cellulare. Se ci trovassimo nel caso in cui nessun evento è avvenuto, sarebbe inutile invocare un nuovo kernel, quindi la computazione dei successivi sotto-alberi viene interrotta.
Siano $A = Alive$, $I = Inactive$, $R = Remove$, il metodo implementato produce i seguenti risultati




(A) Senza bounding (B) Bounding

FIGURA 2.9

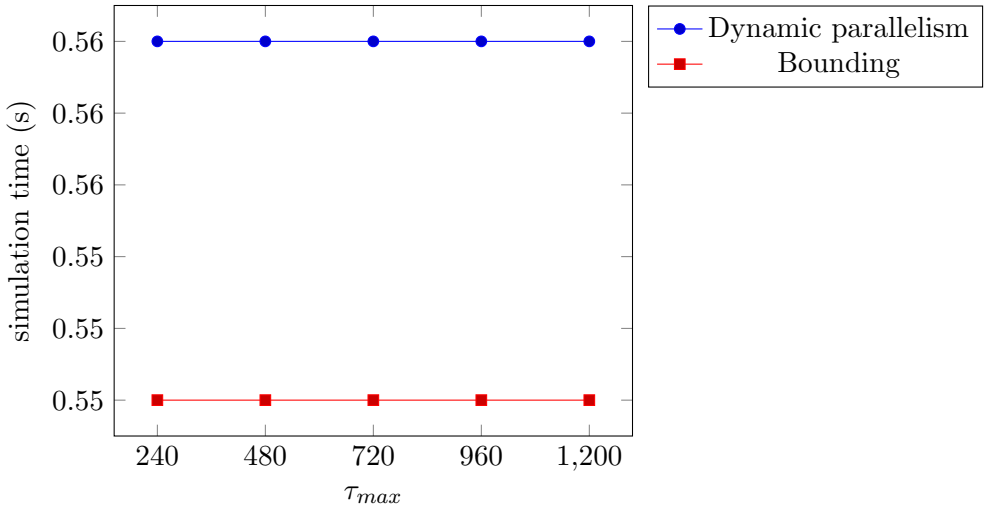



FIGURA 2.10

Il leggero aumento delle prestazioni mostrato in figura indica che la tecnica del *bounding* ha poco impatto sull'aumento delle performance, ma sicuramente contribuisce alla migliore gestione delle risorse computazionali relative alla GPU e all'uso intensivo 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'ultimo accelerante.

**Inferenza del livello di profondità** Sebbene CUDA ci ponga un livello massimo di nesting pari a 24 livelli, nei modelli presi in considerazione per queste simulazioni, difficilmente si supera la soglia dei 5 eventi di proliferazione per ogni cellula, però per come è stato pensato l'utilizzo del parallelismo dinamico vengono allocati $N$ livelli pari al numero massimo di cellule che è possibile mantenere sulla memoria della GPU, dunque se in memoria centrale è presente sufficiente spazio per allocare 10 livelli di profondità essi vengono allocati, consumando più risorse del necessario. Non è nemmeno possibile utilizzare la memoria di *swap* dato che le schede 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 è quello di dare la possibilità all'utente di specificare il numero di livelli da utilizzare durante la simulazione, se è al corrente del numero media di divisioni che ogni cellula andrà ad affrontare. Un altro modo è quello di inferire il livello 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 $\tau$ 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 possiamo stimare che il numero $\eta$ dei livelli di profondità dell'albero è

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

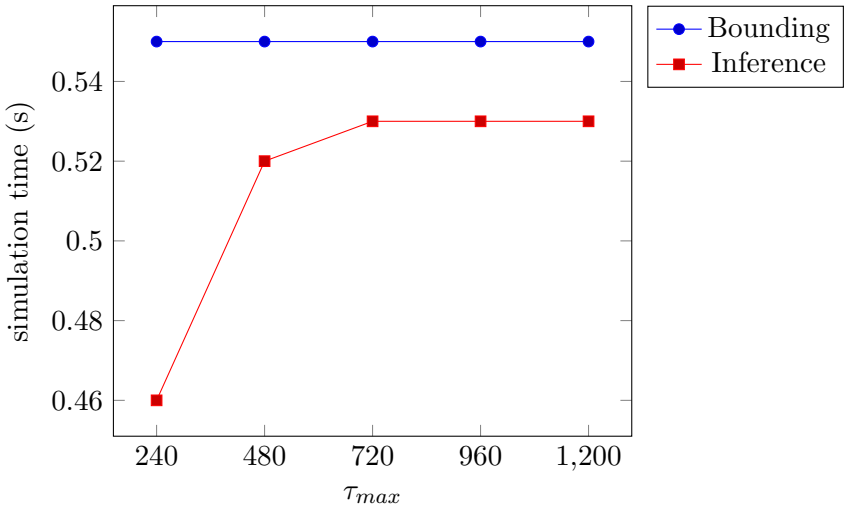



FIGURA 2.11

Nel grafico possiamo notare un maggiore incremento di prestazioni fino ad un valore di $\tau_{max}$ di 720, questo perchè 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. 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 ordinato in modo crescente$\Omega$ 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 $H(0)$ in realtà abbiamo 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, poichè la frequenza riguardante i valori di fluorescenza delle cellule $Inactive$ è già stata aggiornata durante l'esecuzione del kernel. Mediante questa tecnica la $cudaMemcpy()$ viene utilizzata solamente a inizio e fine simulazione per il salvataggio dei risultati.

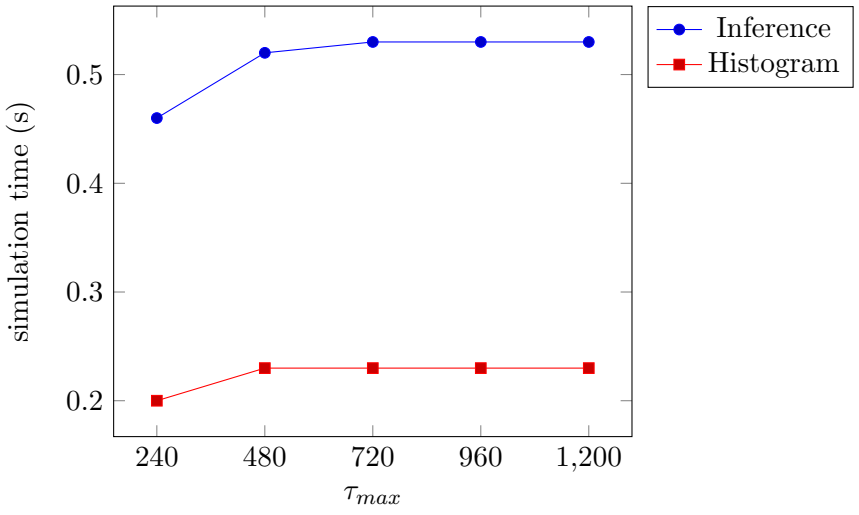



FIGURA 2.12

Come si evince dai grafici, questo è l'accelerante che incrementa maggiormente lo speedup dell'algoritmo, dato che risolve quasi totalmente il problema del bottleneck causato dal trasferimento bidirezionale delle informazioni per lo svolgimento della simulazione.

### 2.3.4 Gestione della memoria

L'implementazione degli acceleranti ha migliorato di gran lunga le performance dell'algoritmo, ma ha introdotto un problema, ovvero la crescita esponenziale della memoria. 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'ultimo accelerante, 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.

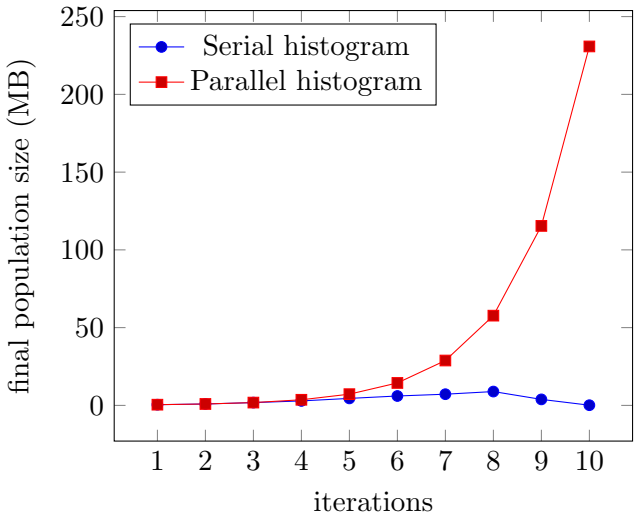



FIGURA 2.13

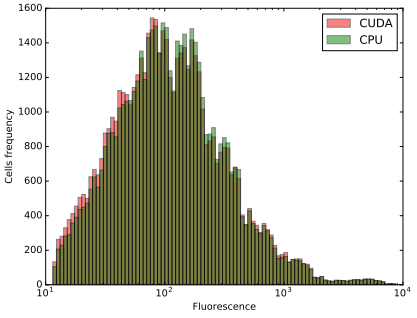
La crescita esponenziale dell'occupazione di memoria potrebbe essere un problema nel momento in cui il tempo $\tau_{max}$ inizia a crescere ulteriormente. La soluzione immediata a questo problema potrebbe essere di natura economica, ovvero l'acquisto di GPU moderne con molta memoria a disposizione. Il 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.

# Capitolo 3

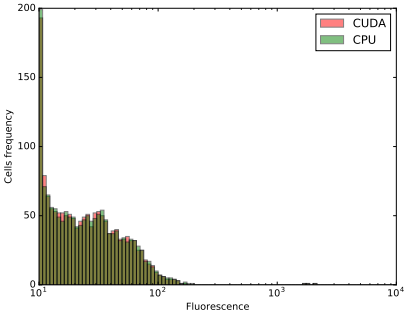
# Risultati

## 3.1 Confronto CUDA/Python

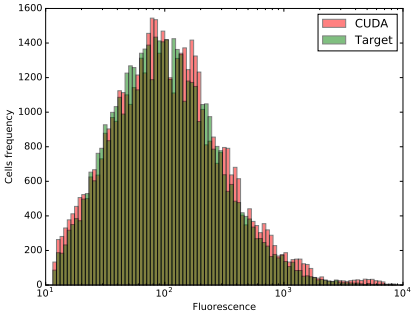
Lo scopo della simulazione è verificare la correttezza del modello utilizzato sulla base di dati reali provenienti da esperimenti effettuati in vivo. Di seguito una comparazione dei risultati ottenuti dalle simulazioni fra l'algoritmo di riferimento sviluppato in Python e la versione implementata tramite CUDA.



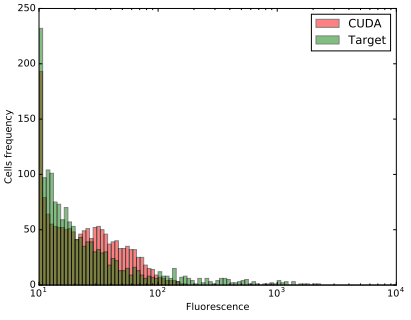

(A) Fitting $\varphi_{min} = 11.0, \tau_{max} = 240$




(B) Validation $\varphi_{min} = 8.7, \tau_{max} = 504$




(C) Fitting $\varphi_{min} = 11.0, \tau_{max} = 240$



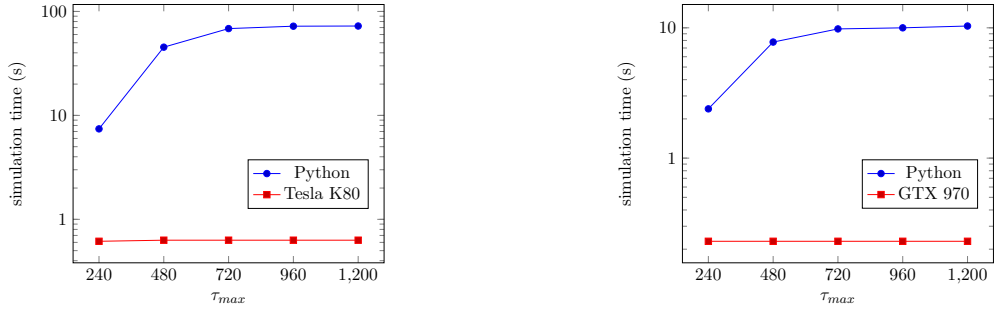

(D) Validation $\varphi_{min} = 8.7, \tau_{max} = 504$

FIGURA 3.1

Come è possibile notare dai grafici in figura, sono presenti aree in cui i valori di frequenza si discostano tra loro di poche 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.

## 3.2 Performance

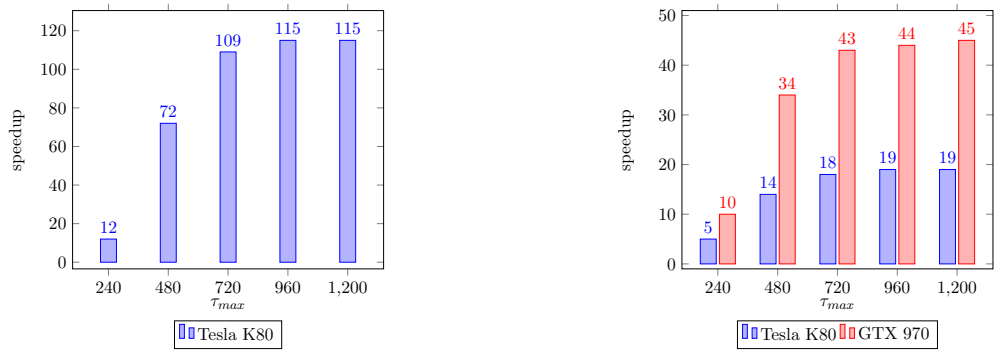
L'implementazione di un modello ad albero per la parallelizzazione di eventi di proliferazione cellulare si è rivelata una tecnica efficiente per questo tipo di problema, di seguito in figura ritroviamo i risultati ottenuti per quanto riguarda i tempi necessari alla computazione della simulazione modificando il parametro $\tau_{max}$.




(A) Fitting $\varphi_{min} = 11.0$ (B) Validation $\varphi_{min} = 8.7$

FIGURA 3.2

Un'idea ancora più generale riguardo allo speedup guadagnato è fornita dal seguente grafico




(A) Fitting $\varphi_{min} = 11.0$ (B) Validation $\varphi_{min} = 8.7$

FIGURA 3.3

Come è possibile notare per quanto riguarda la *validazione*, 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.

# Capitolo 4

# Conclusioni

# Bibliografia


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

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

[3] 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.

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

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