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

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 non riguardanti l'area della computer grafica, ma un ambito più generico, è stato quindi coniato il termine General-Purpose computing on Graphics Processing Units per indicare l'utilizzo di questo tipo di device per applicazioni non legate all'elaborazione delle immagini. 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 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, dato che solitamente si utilizza un cospicuo numero di thread per eseguire un determinato task, altrimenti le prestazioni si abbasserebbero [4].
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 SIMD (Single Instruction, Multiple Data stream)[2] 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.

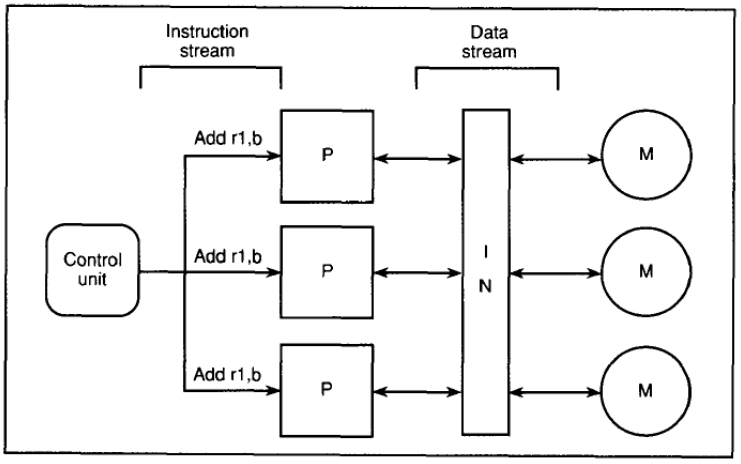


FIGURA 2.1: Schema di esecuzione architettura SIMD[2]

In ambito scientifico l'architetture 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 bit.

Oltre all'architettura SIMD è stata definita anche un'architettura che prevede l'esecuzione di diverse istruzioni su diversi stream di dati, denominata MIMD (Multiple Instructions, Multiple Data streams)[2]. 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, sia essa utilizzando lo scambio di messaggi o la memoria condivisa, dato che le macchine operano in modo asincrono.

Quindi per quanto riguarda l'elaborazione di insiemi fenomeni del mondo reale che obbediscono tutti alle medesime leggi, risulta opportuno utilizzare il paradigma SIMD, dato che è più semplice da implementare e gestire.

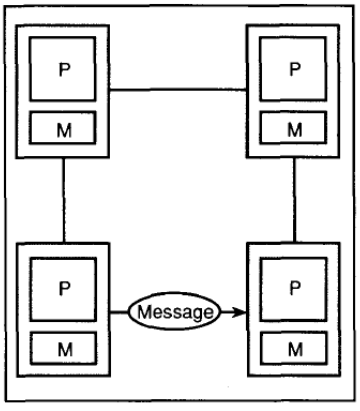


FIGURA 2.2: Schema di esecuzione architettura MIMD[2]

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 poichè 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 ad una frequenza delle unità di elaborazione della GPU molto minore rispetto alla 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 (Graphics Processing Units) per aumentare le prestazioni di alcuni tipi di software. 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) [6]

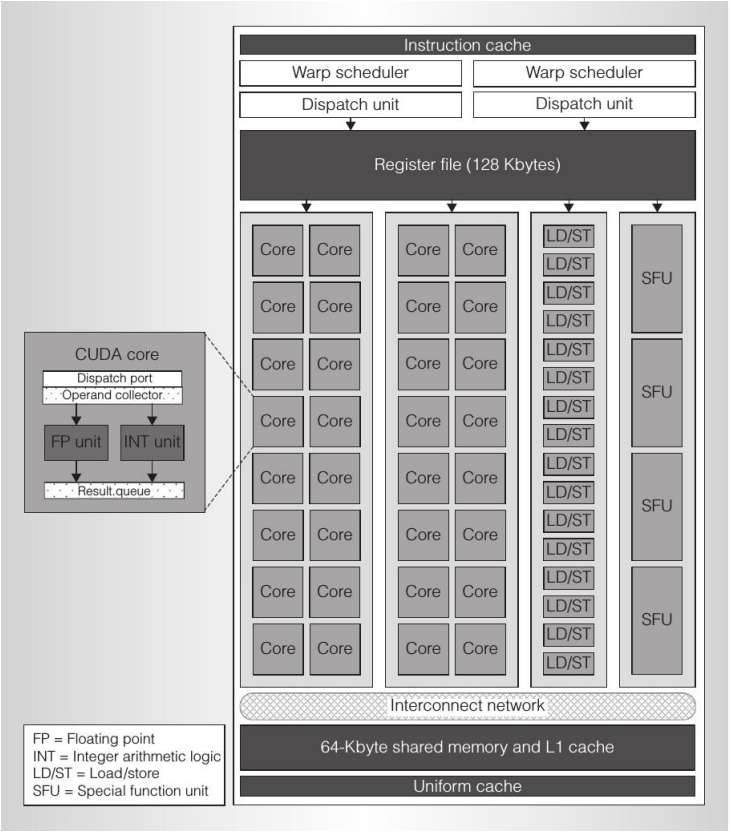


FIGURA 2.3: Streaming Multiprocessor dell'architettura Fermi [6]

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 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 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, ed è quindi obbligatorio procedere alla sincronizzazione di tutti i Thread mediante la funzione *cudaDeviceSynchronize()*.

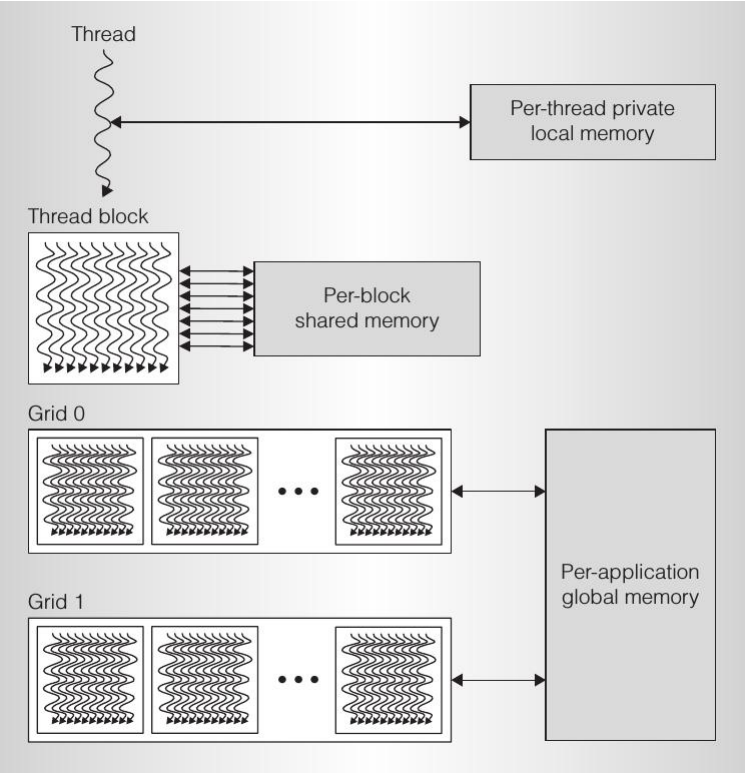


FIGURA 2.4: Schema della gerarchia di Thread, Thread block, e Grid [6]

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, 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, si sta invocando l'esecuzione di un Kernel specificando l'utilizzo di 8 Thread Block e 32 Thread per blocco (quando non specificato, il numero delle Grid da utilizzare ha come valore di default 1). In generale il numero totale di Thread che verranno utilizzati per la computazione del Kernel è

$$\Gamma * B * T$$

dove Γ, B, T sono rispettivamente il numero di Grid, il numero di Thread Block e il numero di Thread che si vuole utilizzare.

La possibilità di modificare la suddivisione del Kernel tra Grid e 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 Grid e Thread Block richiesti durante l'invocazione del Kernel, secondo la seguente relazione:

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

dove Γ, B, T sono rispettivamente il numero di Grid, il numero di Thread Block, e il numero di Thread che devono essere utilizzati e γ, β, τ sono 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 dell'id globale associato ad un Thread durante l'esecuzione di un Kernel

Gerarchie di memoria

Attraverso l'astrazione software CUDA mette a disposizione tre diverse tipologie di memorie[4]:

- 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 all'interno di un Thread.

- Memoria condivisa: la velocità di accesso è pressochè simile a quella dei registri poichè anch'essa è presente all'interno del chip, con l'unica differenza che questa porzione di memoria è accessibile da tutti i Thread appartenenti ad uno stesso Thread Block. Per questo motivo è preferibile utilizzare questo tipo di memoria quando è presente la necessità di salvare dei risultati parziali di una computazione.

- 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 memorizzare i dati di input e output di un determinato Kernel.

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 ad un Thread Block su uno Streaming Multiprocessor (SM)[4]. 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()*.

2.3 Definizione del modello

La simulazione del fenomeno prevede la creazione di una popolazione iniziale di cellule basata su un istogramma fornito in input denominato $H(0)$ contenente coppie di valori (φ_i, ψ_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 all'interno dei campioni analizzati. La popolazione iniziale di cellule è rappresentabile tramite un array di lunghezza pari a

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

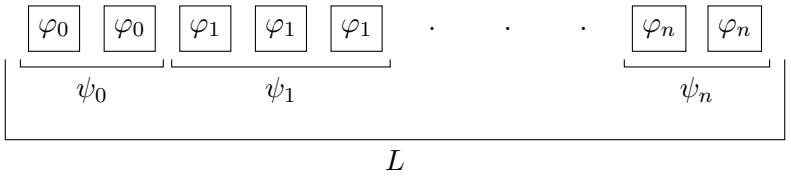


FIGURA 2.5: Rappresentazione della popolazione iniziale di cellule con i rispettivi valori di fluorescenza φ_i tramite un array unidimensionale di lunghezza L

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

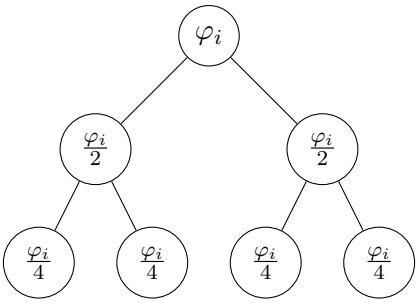


FIGURA 2.6: Fenomeno di divisione cellulare rappresentato tramite albero binario bilanciato

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.

2.4 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 si riduce all'implementazione di un algoritmo in grado di massimizzare il parallelismo necessario alla computazione dell'insieme di array atto a portare a termine la simulazione.

2.4.1 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 sarà stato creato un array i cui elementi saranno le coppie di valori (φ_i, ψ_i). Ora che è nota la frequenza ψ_i per ogni fluorescenza φ_i, è possibile invocare l'esecuzione di un Kernel per ogni coppia di valori tale che al Kernel vengano assegnati esattamente ψ_i Thread e $\frac{\psi_i}{1024}$ Thread Block, in modo tale che ognuno di essi si occupi di generare una cellula avente φ_i valore di fluorescenza.

FIGURA 2.7: Esecuzione di un Kernel formato da ψ_i Thread necessari alla creazione di esattamente ψ_i cellule aventi φ_i come valore di fluorescenza

2.4.2 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 per ogni cellula originaria.
Per la simulazione viene invocato un Kkernel specifico. Sia $X_{0,i}$ un elemento dell'array iniziale con $i < L$, a cui è associato un certo valore φ_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 si occuperà di creare due nuove cellule con fluorescenza dimezzata rispetto alla fluorescenza della cellula originaria e genererà il nuovo timer delle cellule sulla base del tipo di cellula considerata.
Le nuove cellule sviluppate dalla proliferazione di $X_{0,i}$ saranno

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

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

Esse avranno il proprio tempo di vita globale τ 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 τ_{max} e una soglia minima φ_{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 σ l'array degli stati di ogni cellula, allora è possibile 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 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 possiedono un valore di fluorescenza φ sotto la soglia φ_{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 τ_{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.

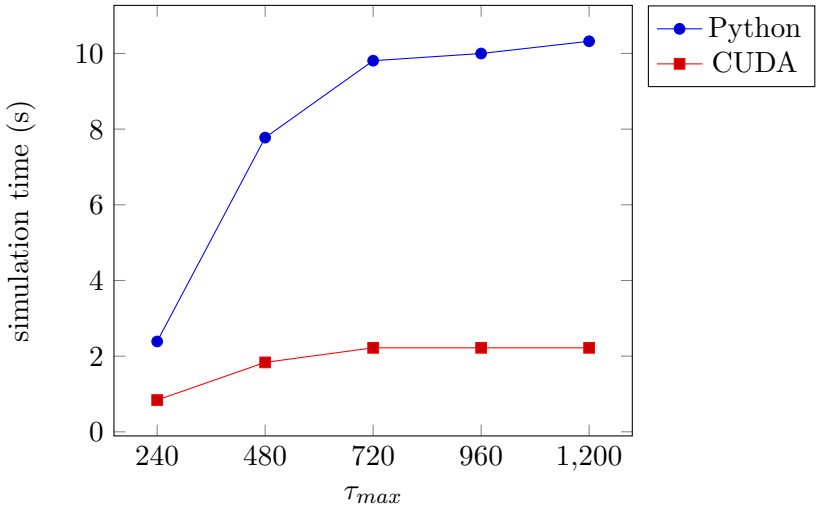


FIGURA 2.8: Confronto dei tempi di esecuzione (in secondi) dell'algoritmo per la simulazione fra la versione sviluppata in Python e quella sviluppata in CUDA C, utilizzando differenti valori di tempo massimo di proliferazione τ_{max}

Come è possibile notare dal grafico 2.8, è presente un cospicuo aumento delle performance rispetto alla versione non parallela dell'algoritmo, ma non è ancora degno di nota per quanto riguarda i risultati che si vogliono ottenere.

2.4.3 Acceleranti

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 alcuni acceleranti 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*[3] in quanto permette di computare strutture intrinsecamente ricorsive, come nel caso del *Radix sort*[5].
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()*[3]. 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.

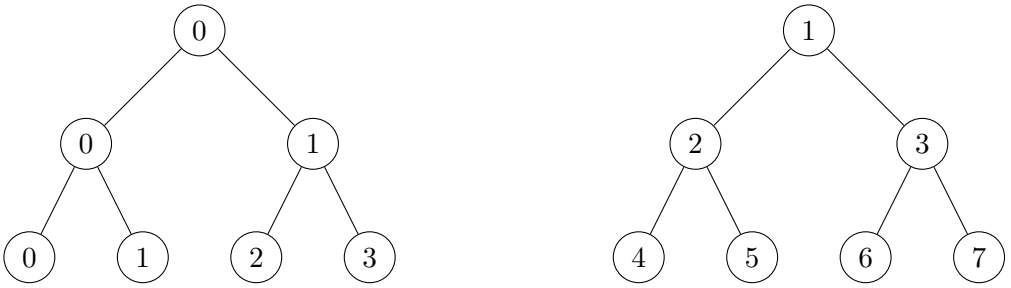


FIGURA 2.9: Rappresentazione di due alberi di proliferazione, le cui radici rappresentano le cellule appartenenti alla popolazione iniziale X_0 mentre ogni livello comprende nodi raffiguranti cellule appartenenti a nuove popolazioni cellulari

In figura 2.9 sono raffigurati 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:

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

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 ino a raggiungere il termine della simulazione.

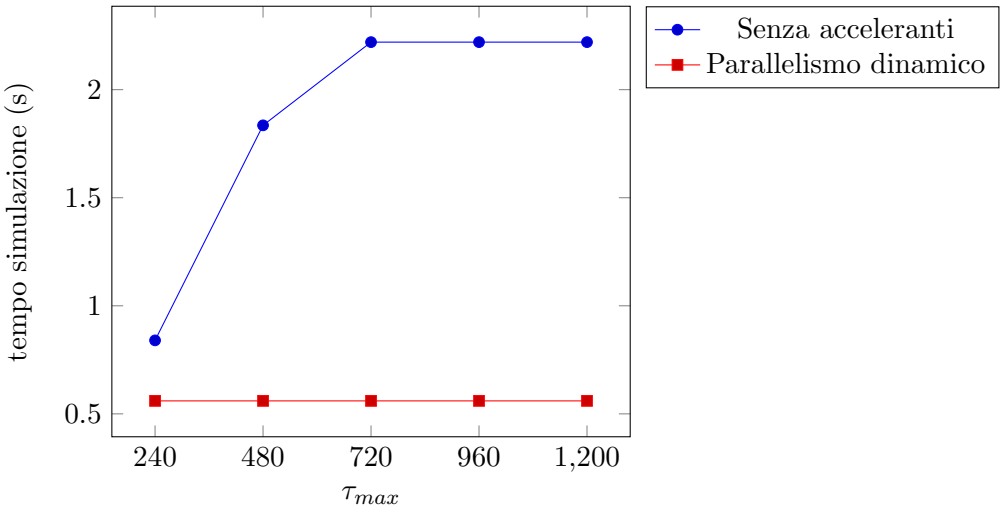
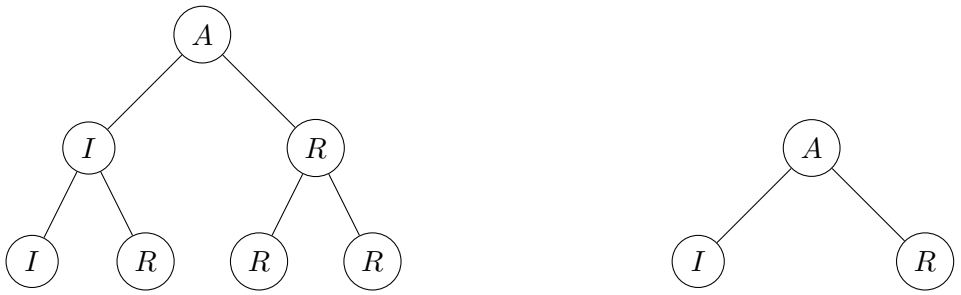


FIGURA 2.10: Confronto dei tempi di esecuzione (in secondi) dell'algoritmo parallelo fra la prima versione senza acceleranti e la versione che implementa il parallelismo dinamico, utilizzando differenti valori di tempo massimo di proliferazione τ_{max}

Dal grafico in figura 2.10 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[7] , ovvero una zona di memoria 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 seguenti risultati



(A) Senza bounding (B) Bounding

FIGURA 2.11: Confronto di alberi di proliferazione cellulare utilizzando il metodo del bounding

Come è possibile notare, l'albero risultante dall'applicazione del bounding presenta meno nodi, ciò equivale al fatto di risparmiare risorse evitando la computazione di sotto-alberi inutili ai fini della simulazione.

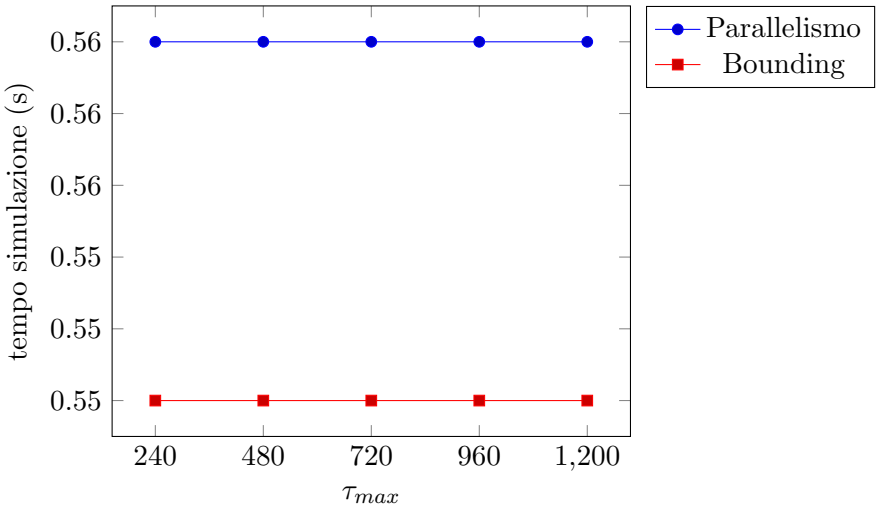


FIGURA 2.12: Confronto dei tempi di esecuzione (in secondi) dell'algoritmo parallelo fra la versione che implementa solamente il parallelismo dinamico e la versione che oltre a quest'ultimo fa uso anche del metodo del bounding, utilizzando differenti valori di tempo massimo di proliferazione τ_{max}

In entrambe la versioni dell'algoritmo mostrate nel grafico in figura 2.12 si nota che il tempo di simulazione rimane costante anche al crescere del limite τ_{max} grazie al parallelismo dinamico implementato in precedenza. Il leggero aumento delle prestazioni mostrato in figura 2.12 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 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 π dei vari tipi di cellule τ che è possibile avere all'interno della popolazione iniziale. Ogni tipo di cellula possiede un tempo medio μ dopo il quale avviene la divisione, il quale segue una distribuzione normale. È possibile pensare di utilizzare il tempo medio μ_i del tipo di cellula τ_i che ha probabilità π_i maggiore per calcolare un numero approssimativo di livello di divisioni tali per cui venga superato il τ_{max} definito per la simulazione.
Sia μ_{max} il tempo medio di divisione del tipo di cellula avente $max(\pi)$. Allora è possibile stimare che il numero η dei livelli di profondità dell'albero è

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

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 η non rispecchi a pieno la realtà che si vuole simulare.

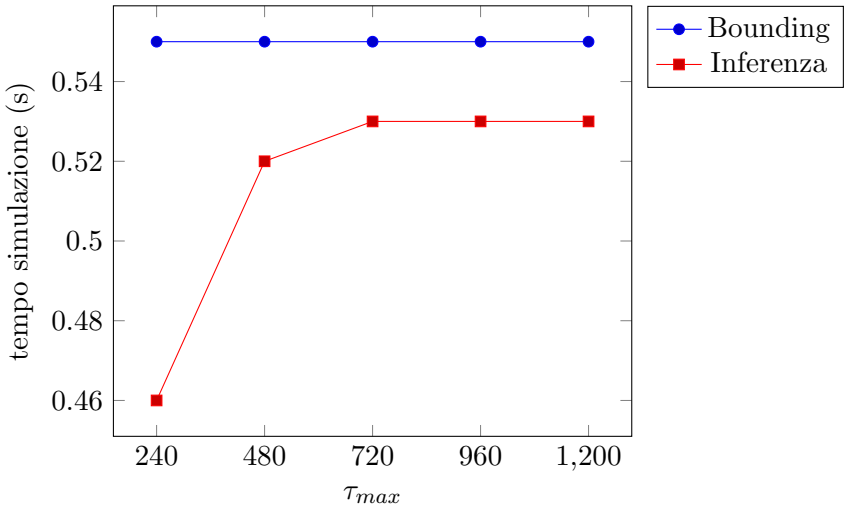


FIGURA 2.13: Confronto dei tempi di esecuzione (in secondi) dell'algoritmo parallelo fra la versione che implementa gli acceleranti descritti fino al metodo del bounding e la versione che oltre a quest'ultimo utilizza anche l'inferenza del numero di livelli necessari alla simulazione, utilizzando differenti valori di tempo massimo di proliferazione τ_{max}

Nel grafico in figura 2.13 si può notare un maggiore incremento di prestazioni fino ad un valore di τ_{max} di 720, questo perché sebbene venga applicata l'inferenza dei parametri, l'algoritmo prevede l'istanziamento di η 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 poichè la GPU utilizzata non ha abbastanza memoria per istanziare un numero di livelli pari a η 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 ordinato in modo crescenteΩ contenente tutti possibili valori di fluorescenza φ 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 Ω e aggiornare il valore di frequenza ψ 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 Ω conterrà tutti i valori di $H(0)$ tali che $\varphi_i \geqslant \varphi_{min}$ e i successivi valori φ_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 del kernel. Mediante questa tecnica la *cudaMemcpy*() viene utilizzata solamente a inizio e fine simulazione per il salvataggio dei risultati.

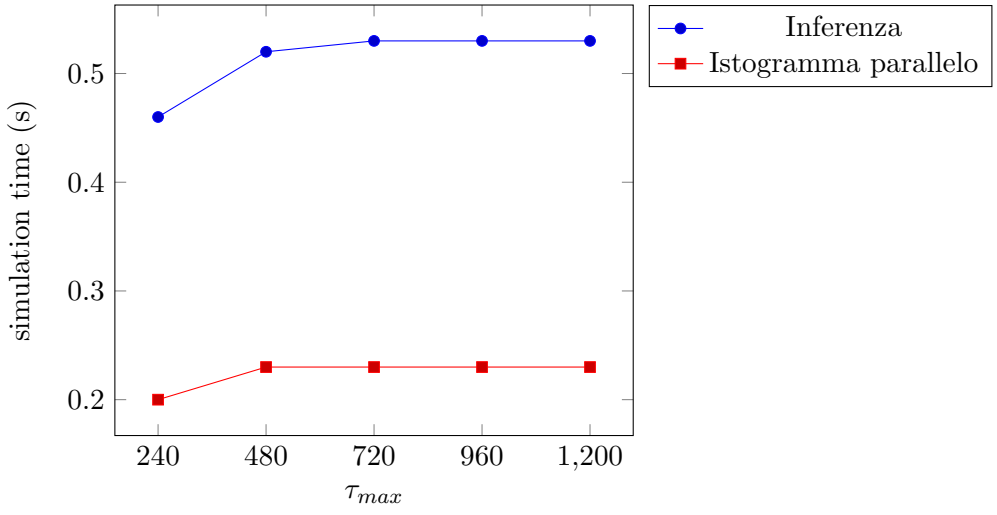


FIGURA 2.14: Confronto dei tempi di esecuzione (in secondi) dell'algoritmo parallelo fra la versione che implementa gli acceleranti descritti fino al metodo dell'inferenza dei parametri e la versione che oltre a quest'ultimo implementa anche il calcolo parallelo dell'istogramma finale, utilizzando differenti valori di tempo massimo di proliferazione τ_{max}

Come si evince dal grafico in figura 2.14, questo è l'accelerante 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.

2.4.4 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.

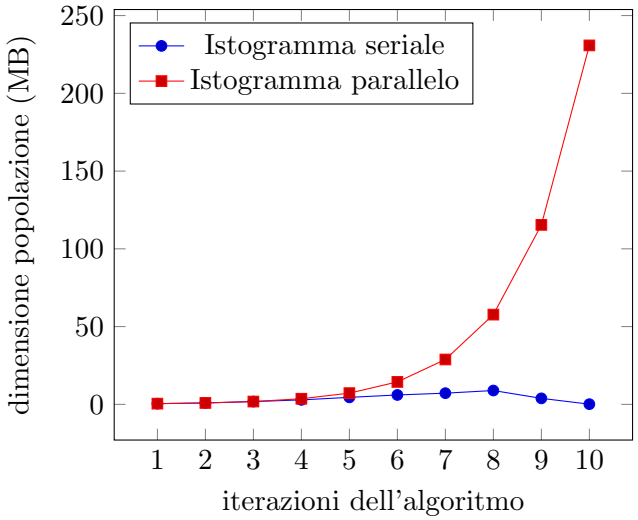


FIGURA 2.15: Confronto dell'utilizzo di memoria fra la versione dell'algoritmo che prevede il calcolo seriale dell'istogramma con rimozione delle cellule non *Alive* e il calcolo parallelo dell'istogramma dei risultati senza filtraggio della popolazione cellulare al termine di n iterazioni dell'algoritmo

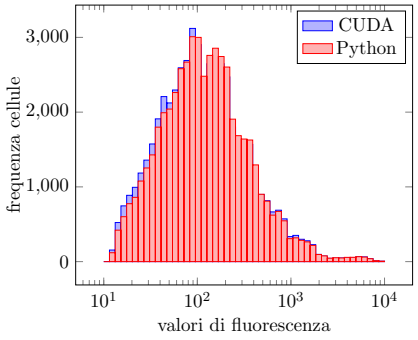
La crescita esponenziale dell'occupazione di memoria visibile in figura 2.15 potrebbe essere un problema nel momento in cui il tempo τ_{max} inizia a crescere ulteriormente. La soluzione immediata a questo problema potrebbe essere di natura economica, ovvero l'acquisto di GPU moderne con molta memoria 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.

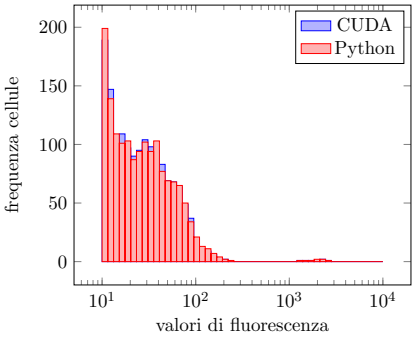
Capitolo 3

Risultati

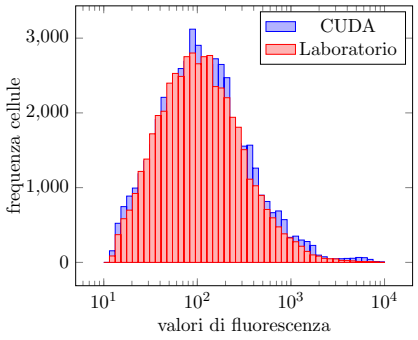
3.1 Confronto CUDA/Python



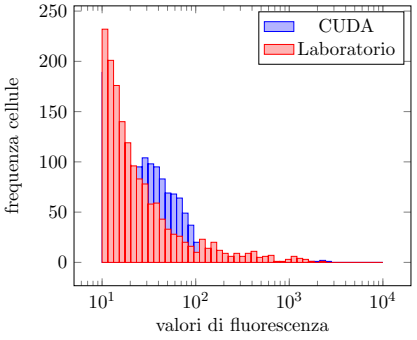
(A) Confronto simulazioni Python e CUDA con $\varphi_{min} = 11.0, \tau_{max} = 240$



(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

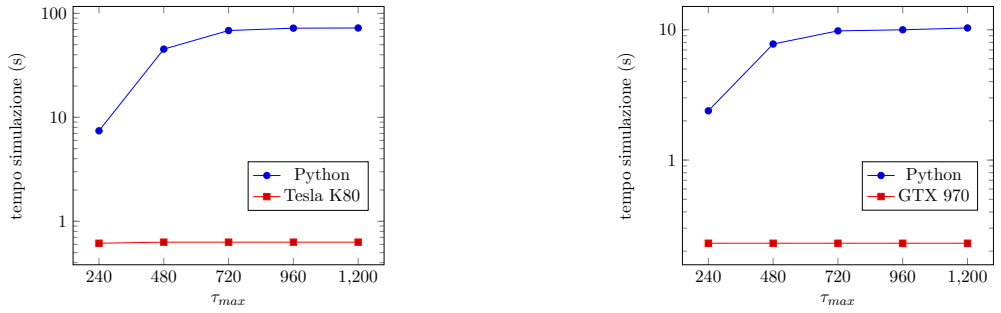
FIGURA 3.1: Confronto dei valori di fluorescenza sperimentali ricavati in laboratorio e quelli ottenuti dalle simulazioni utilizzando l'algoritmo sviluppato in Python e in CUDA con diversi parametri iniziali

Lo scopo della simulazione è verificare la correttezza del modello utilizzato sulla base di dati reali provenienti da esperimenti effettuati in vivo in laboratorio. Di seguito una comparazione

dei risultati ottenuti dalle simulazioni fra l'algoritmo di riferimento sviluppato in Python e la versione implementata tramite CUDA.
In figura 3.1 è possibile vedere la comparazione dei risultati ottenuti dalle simulatzioni 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 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.

3.2 Performance

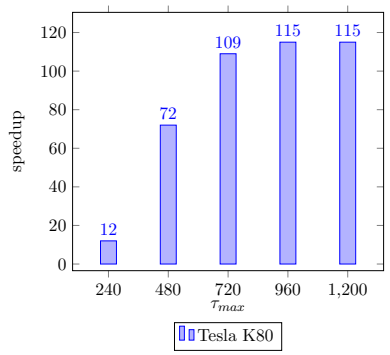
L'implementazione di un modello ad albero per la parallelizzazione di eventi di proliferazione cellulare si è rivelata una tecnica efficiente per questo tipo di simulazione, di seguito in figura sono presenti i risultati ottenuti per quanto riguarda i tempi necessari alla computazione della simulazione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



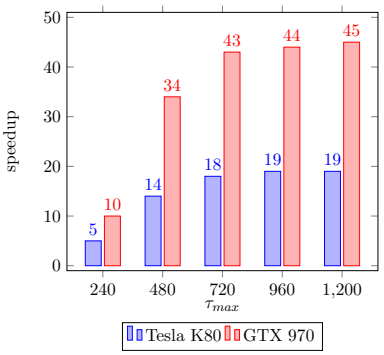
(A) $\varphi_{min} = 11.0$ (B) $\varphi_{min} = 8.7$

FIGURA 3.2: 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 φ_{min} per la simulazione, aumentando il tempo massimo τ_{max} per la proliferazione cellulare

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



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



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

FIGURA 3.3: 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

Come è possibile notare dal grafico 3.3 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 φ_i all'interno dell'array delle frequenze Ω. 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] Ralph Duncan. "A survey of parallel computer architectures". In: *Computer* 23.2 (1990), pp. 5–16.

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

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

[5] 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.

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

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