UNIVERSITÀ
DEGLI STUDI
DI TORINO

Relazione del progetto di Sistemi di Calcolo Parallelo e Distribuito

Santina Lorenzo
matr: 797890

e

Sorrentino Luca
matr: 797180

Link a Github

Indice

Introduzione:

Game of life è uno dei più famosi esempi di *automa cellulare,* ovvero un mondo costituito da spazi omogenei divisi in celle elementari, ognuna delle quali, ad intervalli regolari, cambia stato secondo regole di evoluzione che riguardano la cella stessa e quelle ad essa confinanti. Ad ognuna delle celle viene associato un numero finito di stati, ad esempio viva oppure morta. Lo stato di una cella, non dipende solo dalla cella stessa, ma anche da quelle sue confinanti.

Regole del gioco:

Il mondo si evolve per iterazioni, e di volta in volta lo stato delle celle viene definito in base a 4 semplici regole:

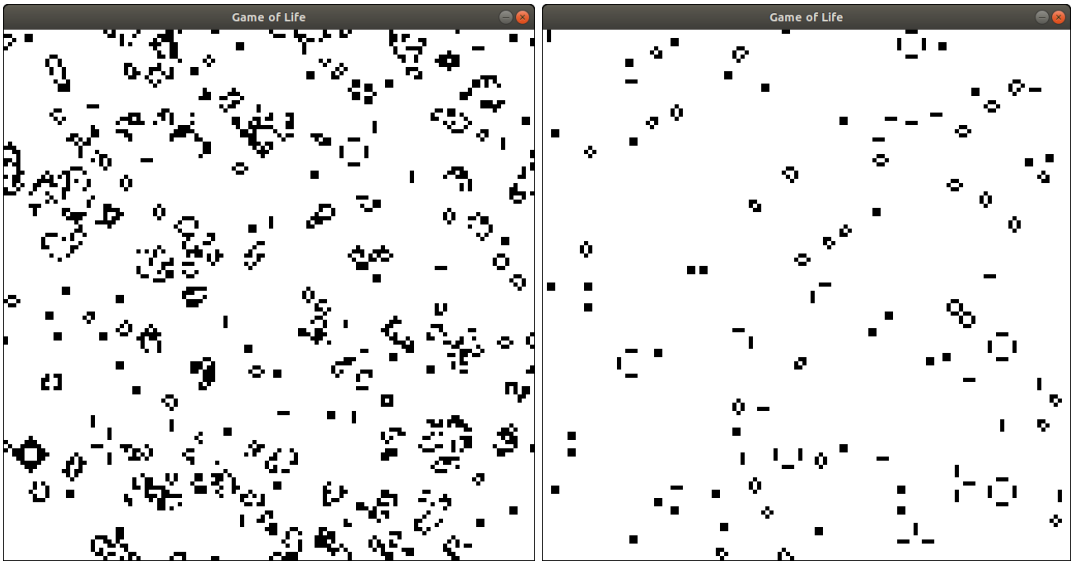
1. Ogni cellula viva con meno di 2 vicini, muore a causa dell' isolamento
2. Ogni cellula viva con esattamente 2 o 3 vicini, resta in vita nella prossima generazione
3. Ogni cellula viva con più di 3 vicini, muore a causa del sovraffollamento
4. Ogni cellula morta con esattamente 3 vicini si riattiva nella prossima iterazione, come se si fossero riprodotte

Un modo per riassumere queste regole, in modo da dover computare meno condizioni, è quello di dire che nella prossima iterazione la cellula sarà attiva solo se ha esattamente 3 vicini (in questo caso è indifferente se la cellula in questione sia viva o morta) o se ne ha due ma attualmente è viva. In questo modo, tutte le condizioni possono essere verificate in una unica riga di codice (dopo aver contato il numero di cellule vicine attualmente vive):

```
(aliveCells == 3 || (aliveCells == 2 && cell) ? 1 : 0;
```

Di seguito è possibile vedere un'istanza di game of life, lanciata sulla nostra interfaccia grafica.

Sulla sinistra abbiamo uno screen catturato dopo poche iterazioni, sulla destra uno screen catturato quando ormai il mondo ha raggiunto una situazione di stabilità.



Struttura del codice:

Ciò che volevamo ottenere attraverso questo progetto, era osservare l'incremento delle prestazioni del programma a seguito dell'implementazione parallela e di varie ottimizzazioni.

A tal proposito abbiamo effettuato prima un'implementazione sequenziale, poi una multithread mediante OpenMP e infine una con GPGPU usando Cuda.

Struttura fisica:

Nonostante le diverse implementazioni, siamo riusciti a mantenere un unico eseguibile, più comodo per chi utilizza il software ma che ha reso la creazione del makefile più complessa.
Infatti nvcc non supporta openmp, inoltre i file hanno estensioni diverse e andavano specificate indicazioni al compilatore apposite per le librerie dell'interfaccia grafica.
E' stato inoltre necessario usare il compilatore di c++ perchè il linker di nvcc andava in conflitto con quello di c.

La cartella del progetto contiene quindi 4 elementi:

- **makefile**: esegue prima la compilazione del file cuda con nvcc e poi lo linka nell'eseguibile finale compilato con g++.
  Sono stati attivati i flag di ottimizzazione;
- **bin**: in questa cartella si ritroverà l'eseguibile al termine della compilazione;
- **source**: contiene tutti i sorgenti;
- **header**: contiene i rispettivi header dei sorgenti.

Per compilare basta quindi recarsi nella cartella principale e lanciare il comando make.

Una volta compilato, il programma può essere lanciato con uno dei tre seguenti parametri:

- **gui**: permette di visualizzare a schermo l'evolversi del mondo nelle varie iterazioni;
- **bench**: permette di raccogliere dati sui tempi di esecuzione delle diverse implementazioni;
- **test**: permette di lanciare gli unit test e verificare il corretto funzionamento del programma.

Struttura logica:

Implementare più volte lo stesso algoritmo, se non strutturato bene, può portare alla duplicazione del codice.

Per esempio, ogni implementazione necessita di una classe di unit test da importare per verificare i risultati ottenuti, ma ciò avrebbe comportato una duplicazioni di routine di testing identiche (lo stesso vale per le routine dei benchmark).

Per questo motivo abbiamo strutturato il codice in modo tale che tutta la differenza tra le varie implementazioni fosse limitata alla singola funzione responsabile di computare l'iterazione sul mondo.

Questo ci ha portato a creare il file (common) che racchiude tutte le funzioni comuni e la dichiarazione della struct che usiamo per descrivere lo stato del mondo.

Questo file verrà poi incluso dalle tre classi principali (sequential, openmp, cuda) le quali a loro volta espongono la funzione che prende una generazione in input e restituisce la generazione successiva. Il file cuda in realtà esegue anche delle operazioni un pò più complesse (ma comunque trasparenti a tutte le altre classi esterne), in quanto deve gestire il trasferimento di memoria da host a

device, e interroga il device per poterne sfruttare a pieno le caratteristiche hardware.

In questo modo anche il problema con gli altri file (unit_test e benchmark) è risolto, in quanto è stato possibile creare un'unica funzione in cui la routine da usare per il calcolo dell'iterazione successiva viene fornita come parametro di input. In particolare nel codice della gui, tale funzione viene scelta dall'utente tramite parametro.

Esempio di codice all'interno del file benchmark:

```
for (uint32_t world_size = 32; world_size <= 4096; world_size = world_size*2){
        printf("\nWorld Size: %d \n", world_size);
        printf("(Benchmark Cpu) -> %lf\n", get_execution_time(world_size, world_size, iterations, (&seq_compute_generations)));
        printf("(Benchmark OpenMp) -> %lf\n", get_execution_time(world_size, world_size, iterations, (&omp_compute_generations)));
        printf("(Benchmark Gpu) -> %lf\n", get_execution_time(world_size, world_size, iterations, (&compute_cpu_generations_on_gpu)));
}
```

Infine vi è il file game_of_life che si occupa di leggere e parsificare i comandi dell'utente per poi lanciare l'esecuzione di unit test o dei benchmark o dell'interfaccia grafica.

Gui:

Per mostrare il susseguirsi delle generazioni a video, viene fatta eseguire una iterazione dell'algoritmo alla volta e il risultato viene stampato con la libreria SDL2.
La superficie della finestra è inizialmente tutta bianca ed è suddivisa in una griglia.
Si scorre tutta la matrice con il risultato dell'iterazione e se il valore alle coordinate attuali è 1, allora si colora il quadrato di nero, altrimenti lo si lascia bianco.

Quindi:

- celle bianche → morte
- celle nere → vive

Durante la proiezione dell' iterazione attuale, si aspetta un piccolo delay prima di eseguire la successiva per dar modo all'occhio umano di percepire l'immagine.

E' possibile scegliere l'implementazione dell'algoritmo di computazione della generazione fra:

- CPU sequenziale
- CPU parallelo (OpenMP)
- GPU (CUDA)

Sequenziale:

Sono state provate diverse implementazioni con l'obiettivo di eseguire sempre meno calcoli per ogni cella.

Vengono qui di seguito elencate dalla peggiore alla migliore:

- Singlefor:

  Un singolo ciclo for scorre l'intera matrice come se fosse un array (monodimensionale).

  Cosi facendo e' necessario ricalcolare le coordinate degli 8 adiacenti per ogni cella.

- Doublefor:

  La matrice viene acceduta nel classico modo righe e colonne.

  Questo permette di risparmiare parecchi calcoli dato che le coordinate delle righe saranno calcolate solo y volte e non piu x*y volte.

- Pow2:

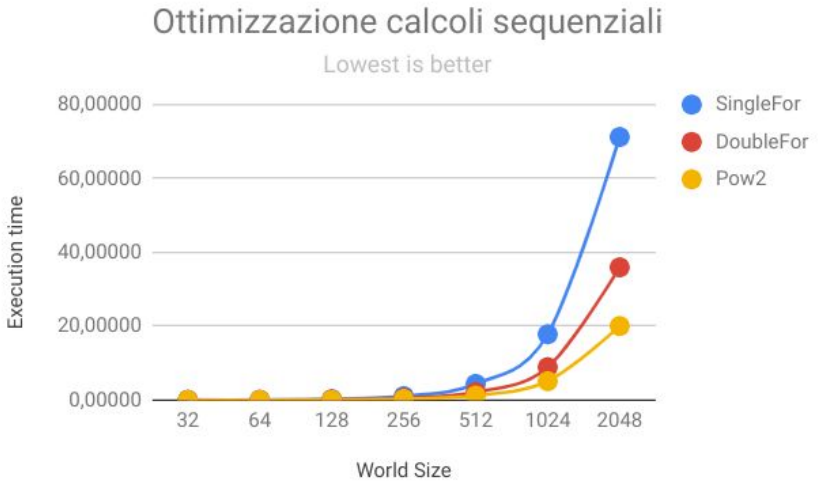
  Nonostante le CPU abbiano ormai tecniche per eseguire le divisioni e i moduli velocemente, se si evita di effettuarle si possono aumentare comunque le prestazioni.

  Inoltre sulle GPU l'esecuzione di un modulo è molto costosa, quindi in ottica della successiva

implementazione in CUDA abbiamo cercato delle soluzioni.

Se il divisore è una potenza di due, è sufficiente effettuare la bitmask del dividendo con il divisore-1. In questo modo si otterrà lo stesso risultato che si otteneva effettuando l'operazione di modulo. Questo implica però che la matrice abbia come dimensioni delle potenze di 2.

World Size	SingleFor	DoubleFor	Pow2
32*32	0,08387	0,01595	0,00828
64*64	0,07861	0,03409	0,01996
128*128	0,27220	0,13816	0,07824
256*256	1,08421	0,55570	0,30077
512*512	4,38697	2,16172	1,23340
1024*1024	17,83341	8,91461	5,13104
2048*2048	71,24473	35,95383	20,04731



OpenMP:

Per distribuire la computazione di ogni generazione su tutti i core della CPU abbiamo parallelizzato il for esterno con OpenMP.
In questo modo ogni core eseguirà la computazione su un sottoinsieme di righe della matrice.
E' una forma di LoadBalancing statica, dato che si assegna a priori il carico di lavoro a ogni core.
Sono state usate due matrici (una di input e una di output) in modo da non dover aspettare che tutti i vicini avessero letto la propria cella prima di poterla aggiornare.
In questo modo è sufficiente la barrier implicita di OpenMP al fondo del for esterno per attendere che tutte le celle siano state aggiornate prima di proseguire all'iterazione successiva.
Alla fine di ogni iterazione le due matrici vengono scambiate così che siano sufficienti due sole matrici per qualsiasi numero di iterazioni si voglia eseguire.

CUDA:

Sono stati provati diversi approcci per testare le performance in funzione del bilanciamento di carico e dell'accesso alla memoria. Si tiene inoltre conto delle caratteristiche del Device in modo da ottimizzare l'esecuzione anche in funzione dello specifico hardware.
Per sfruttare al massimo il parallelismo vengono usati blocchi di dimensione uguale al numero di core per StreamMultiprocessor. In questo modo se ci fosse un numero di blocchi uguale al numero di SM, tutti i thread avrebbero la possibilità di essere eseguiti contemporaneamente.

- Versione monodimensionale

  In questa versione la matrice viene acceduta come fosse un array, gli indici delle celle adiacenti sono calcolati in modo esplicito in base all'id del thread.

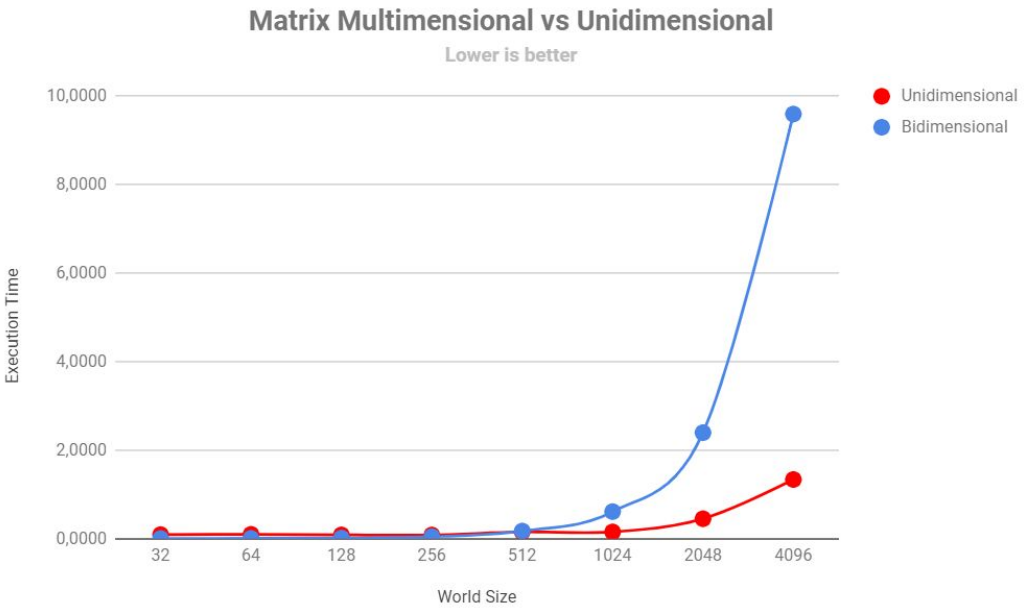
  - Nel caso in cui il numero di celle sia minore o uguale alla dimensione di un blocco, tutte le interazioni possono essere eseguite senza mai uscire dal kernel, sincronizzando i thread del blocco con una chiamata a _syncthread.
  - Nel caso in cui il numero di celle superi la dimensione di un blocco, è ovviamente necessario usare più blocchi.

Per poterli sincronizzare alla fine ogni iterazione è necessario però uscire dal kernel e rientrarci.

- Versione bidimensionale

In questa versione la matrice viene acceduta con gli indici di riga e colonna, ogni blocco ha due dimensioni e in questo modo è più facile indicizzare le celle adiacenti ma al costo di far eseguire più operazione implicite alla gpu e questo è il motivo per cui in questa versione le performance peggiorano.

World Size	Unidimensional	Bidimensional
32*32	0,0931	0,0053
64*64	0,0998	0,0083
128*128	0,0865	0,0146
256*256	0,0847	0,0464
512*512	0,1542	0,1739
1024*1024	0,1542	0,6133
2048*2048	0,4518	2,3976
4096*4096	1,3390	9,5987



- Shared memory

  Per ridurre il numero di accessi alla global memory in ogni blocco ogni thread copia la propria cella nella shared memory, cosi che i successivi accessi da parte dei thread adiacenti siano molto meno costosi.

  E' necessario portare in shared memory anche le celle intorno allo slice da computare, così che le celle sui bordi non debbano accedere alla global memory per prelevare le celle adiacenti.

  Sono state implementate due versioni:

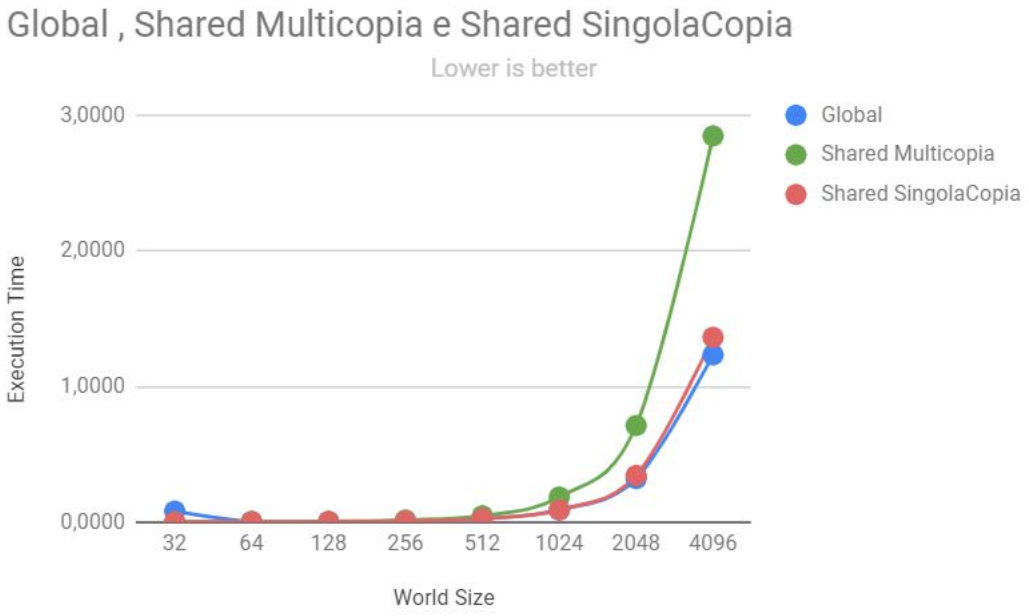
  - Nella prima ogni thread del blocco effettua 3 copie dalla global alla shared (ovvero copia la sua cella più quelle aggiuntive dei bordi), ma questo aumenta

la serializzazione del codice e quindi non si ottengono performance ottimali.

- Nella seconda versione invece, vengono avviati il triplo dei thread, in modo che ogni cella possa essere copiata in parallelo alle altre, e poi solo i thread "centrali" aggiornano il valore della cella.

Purtroppo non c'è stato alcun miglioramento nelle performance rispetto alla versione senza shared memory. Molto probabilmente questo è dovuto alla presenza della cache, che già nella implementazione senza shared memory permetteva di ridurre il costo dei successivi accessi alla memoria globale.

World Size	Global	Shared Multicopia	Shared SingolaCopia
32*32	0,0820	0,0037	0,0035
64*64	0,0040	0,0051	0,0040
128*128	0,0051	0,0059	0,0049
256*256	0,0094	0,0164	0,0096
512*512	0,0269	0,0478	0,0254
1024*1024	0,0906	0,1861	0,0916
2048*2048	0,3224	0,7129	0,3454
4096*4096	1,2336	2,8478	1,3630

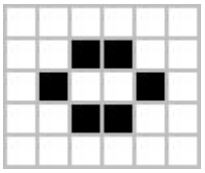


Unit Test:

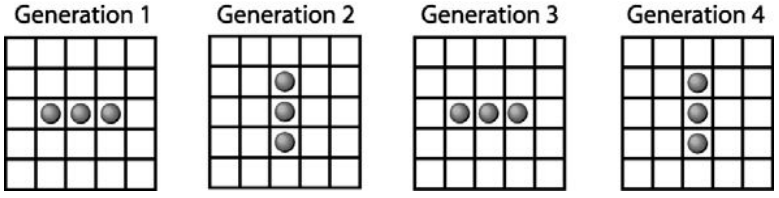
Dato che il calcolo delle varie iterazioni non porta ad un risultato specifico e noto a priori, è di particolare importanza andare a verificare il corretto sviluppo da una iterazione verso la successiva.
Inoltre si rende necessario rieffettuare il controllo ad ogni ottimizzazione dell'algoritmo.

Le regole di base dell'algoritmo sono poche e semplici, per cui abbiamo individuato 3 casi in cui comparivano tutte le possibilità. Vi sono molti esempi di composizioni di celle il cui comportamento è noto a priori ed a cui sono stati assegnati addirittura dei nomi specifici. In particolare noi abbiamo selezionato:

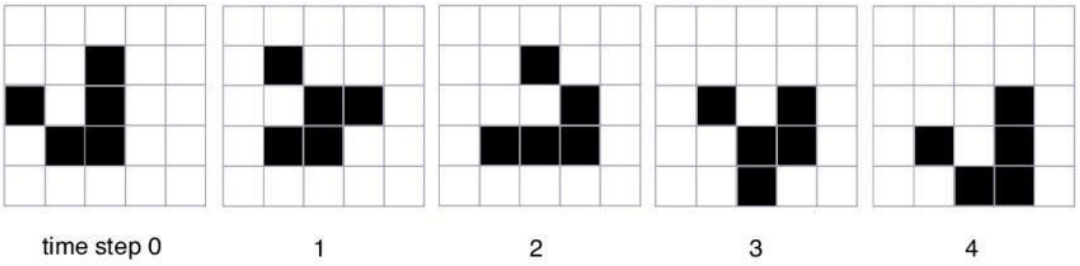
Il **Beehive**: rappresenta il caso più semplice, quello in cui non vi è alcuna differenza tra lo stato in una iterazione e quello nella sua successiva.



Il **Blinker**: in questo caso vi sono sia elementi che "muoiono", sia elementi che "nascono" e sia elementi che non mutano il loro stato. La posizione delle celle però rimane sempre fissa in un punto del mondo e non si ha l'illusione di movimento.



Il **Glider**: l'esempio più completo in cui la figura cambia nel corso di 4 iterazioni, tornando poi nella sua forma originale alla 5 ma traslata verso il basso rispetto al punto di partenza nella prima iterazione. Quest'ultimo esempio ci ha permesso di verificare il corretto funzionamento dei moduli che permettono di collegare la fine del mondo in una direzione con l'inizio dello stesso nella direzione opposta.



Per testare questi 3 casi di base abbiamo creato 3 piccoli mondi ad hoc, in cui abbiamo settato manualmente la posizione iniziale delle varie celle per ricreare quelle delle 3 figure sopracitate.

A questo punto abbiamo fatto eseguire le singole iterazioni alle routine da testare, andando poi di volta in volta a confrontare il risultato ottenuto con quello che ci saremmo aspettati (ovvero quelli mostrati nelle immagini qui sopra).

Mentre nell' implementazione sequenziale ed OpenMP il codice che viene eseguito è sempre lo stesso, indipendentemente dalle dimensione del mondo, per cuda non è lo stesso.
Infatti il kernel da invocare cambia in funzione alla dimensione del mondo. Per questo motivo i test manuali vengono effettuati solo sui kernel dedicati ai mondi di piccole dimensioni.
Per effettuare lo unit test sugli altri kernel abbiamo deciso di eseguire la stessa configurazione di base sia su gpu che su cpu e poi confrontare il risultato ottenuto.
Il test è passato se i risultati sono identici.

Benchmark :

Specifiche del pc su cui sono stati effettuati i test:

I test sono stati effettuati su un pc avente:

- **Processore**: Intel(R) Dual Core(™) i3-6100 da 3.70GHz (4 Thead)
- **Ram**: 8.00 GB
- **Scheda Video**: GeForce GTX 960

Ulteriori dettagli della scheda video:

```
Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 960"
  CUDA Driver Version / Runtime Version          9.1 / 9.1
  CUDA Capability Major/Minor version number:    5.2
  Total amount of global memory:                 2000 MBytes (2097348608 bytes)
  ( 8) Multiprocessors, (128) CUDA Cores/MP:     1024 CUDA Cores
  GPU Max Clock rate:                            1342 MHz (1.34 GHz)
  Memory Clock rate:                             3505 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 1048576 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D=(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.1, CUDA Runtime Version = 9.1, NumDevs = 1
Result = PASS
```

Raccolta dei dati:

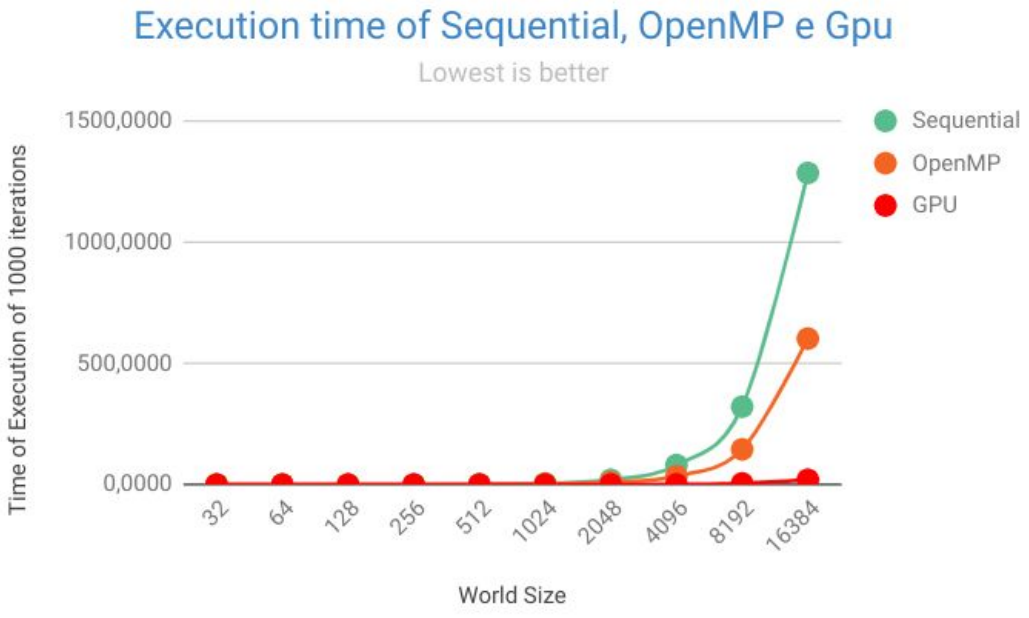
A questo punto, avendo ottimizzato di volta in volta tutte le varie implementazioni, il software è pronto per produrre i dati necessari per la valutazione del grado di parallelismo ottenuto.

Ragionando sulla natura di Game of Life, abbiamo subito notato che si trattava di un problema parzialmente parallelizzabile, ovvero di un problema in cui parte dell'esecuzione deve essere necessariamente svolta in modo sequenziale, perché non è possibile iniziare una nuova iterazione senza prima aver finito la precedente in quanto c'è dipendenza tra i dati di una e l'altra. L'unica cosa che può essere parallelizzata è l'esecuzione all'interno di una iterazione, ed è proprio quella che noi vogliamo analizzare.

Per questo motivo, abbiamo deciso di limitare il numero di iterazioni e piuttosto concentrarci sulla dimensione del mondo, quindi abbiamo bloccato il numero di iterazioni a 1000, abbiamo catturato i tempi di esecuzioni e abbiamo fatto variare di volta in volta la dimensione del mondo.

Seguendo questo principio abbiamo prodotto la seguente tabella con il rispettivo grafico:

EXECUTION TIME FOR 1000 ITERATIONS			
World Size	Sequential	OpenMP	GPU
32*32	0,0055	0,0048	0,0740
64*64	0,0195	0,0113	0,0040
128*128	0,0787	0,0413	0,0046
256*256	0,2984	0,1497	0,0085
512*512	1,2493	0,6183	0,0249
1024*1024	4,9520	2,2360	0,0951
2048*2048	20,0601	9,3372	0,3228
4096*4096	80,6796	34,6138	1,2437
8192*8192	321,3943	145,4929	4,9201
16384*16384	1285,2183	602,2425	19,9971

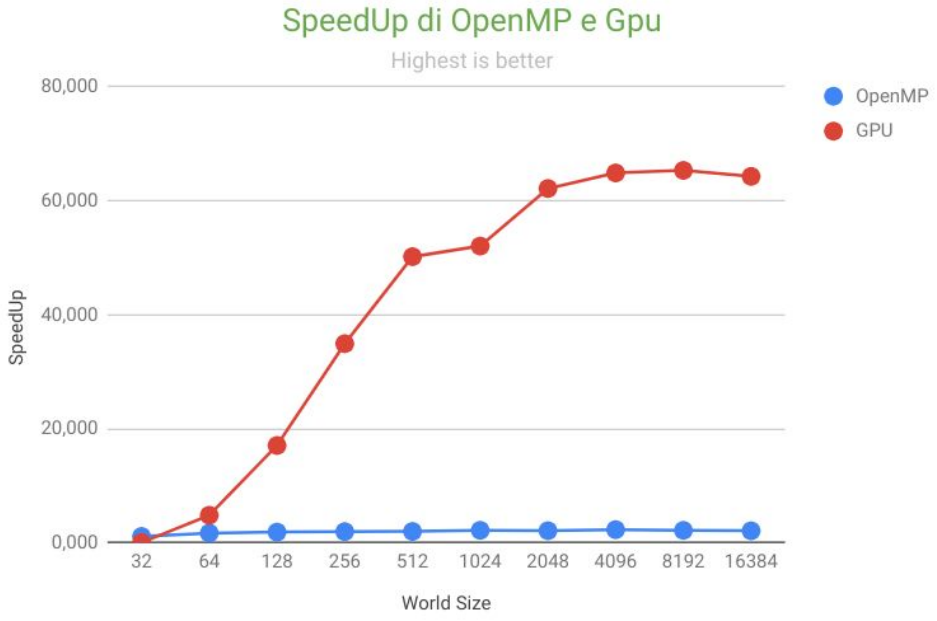


A questo punto possiamo andare a calcolare lo speedup ottenuto (dividendo il tempo di esecuzione dell'implementazione sequenziale per quello ottenuto con le implementazioni OpenMP e Cuda), andando poi a confrontare i risultati ottenuti nella seguente tabella.

	SPEEDUP	
World Size	OpenMP	GPU
32*32	1,148	0,074
64*64	1,730	4,857
128*128	1,906	17,091
256*256	1,993	34,944
512*512	2,020	50,197
1024*1024	2,215	52,057
2048*2048	2,148	62,146
4096*4096	2,331	64,873
8192*8192	2,209	65,323
16384*16384	2,134	64,270

Da questa tabella notiamo che lo speedup cresce linearmente con il crescere del mondo. Nel caso di OpenMp arriva a 2,3 volte, quindi i tempi di esecuzione vengono dimezzati, mentre in Cuda si arriva fino ad uno speedup maggiore di 65 volte, per mondi di dimensione pari a $2^{16} x\, 2^{16}$.

Il rapporto tra i risultati ottenuti è apprezzabile dal seguente grafico



Conclusioni:

Grazie a questo progetto abbiamo avuto modo di comprendere sia le potenzialità che le limitazioni del calcolo parallelo su Cuda. Ad esempio, una delle prime cose che abbiamo verificato è che anche se il numero di core è limitato è comunque preferibile aumentare il numero di thread impiegati piuttosto che far eseguire più operazioni ai singoli thread.

Inoltre dai risultati ottenuti notiamo come la cache esegua già un ottimo lavoro rispetto alla shared memory, infatti se quest'ultima viene usata solo per effettuare letture, non ci sono miglioramenti nelle prestazioni.

Game of Life è un tipico problema di natura Io Bound, ovvero uno in cui le operazioni eseguite sono irrisorie rispetto alla mole di dati che vengono spostate durante le varie iterazioni.

Un modo per incrementare ulteriormente le prestazione quindi, potrebbe essere quello di assegnare ad ogni cella un solo bit, piuttosto che un intero byte, così da non sprecare larghezza di banda.