Progetto di Sistemi di elaborazione accelerata

Sistemi ad elaborazione accelerata

Lorenzo Arlotti - Pierluca Pevere

Anno accademico 2025/2026

Abstract

Le simulazioni di tipo "*falling sand*" rappresentano una classe di automi cellulari in cui ogni elemento di una matrice **simula una particella soggetta a particolari regole** (in questo caso la gravità). Esistono diverse implementazioni, che variano per complessità, per tipi di particelle coinvolti e per regole di interazione tra di esse.
Questa relazione riguarda il progetto di ottimizzazione SIMD e SIMT di una simulazione che prevede l'esistenza di due particelle (oltre a quelle statiche di *vuoto* e *muro*) cioè la *sabbia* e l'*acqua*, svolta per il progetto del corso di "**Sistemi di elaborazione accelerata**".

Lorenzo Arlotti, *Pierluca Pevere*

Indice

1 Introduzione

Esistono diverse implementazioni dell'automa cellulare[1] *falling sand* che prevedono diversi insiemi di regole. Al fine di delineare l'algoritmo più adatto ai nostri scopi, sono state analizzate e confrontate diverse implementazioni, fino a trovare quella considerata migliore per i nostri obiettivi.
Il principale vincolo posto è stato quello di prevedere un'implementazione **completamente deterministica e replicabile in differenti versioni**, in modo da poter ottenere gli stessi risultati in output a partire dallo stesso stato iniziale.
L'obiettivo di ogni implementazione è la generazione dello stato $n + 1$ una volta fornito lo stato n
In particolare sono stati utilizzati i seguenti stati di input:

- il *sample-1* è una matrice 400×400 contenente una massa di *sabbia*

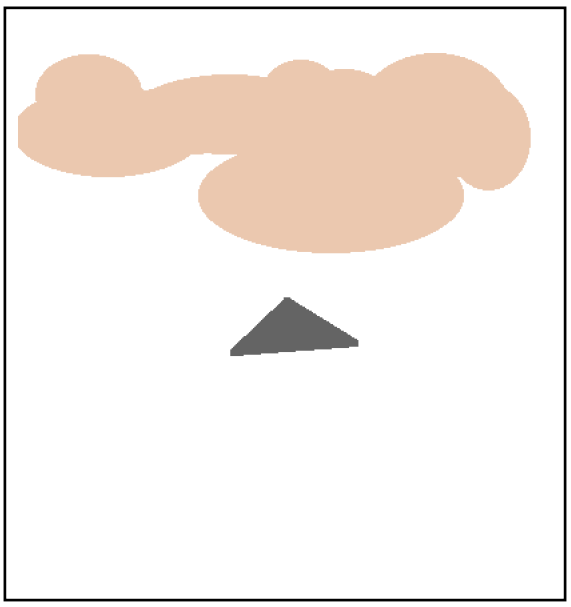

Figura 1: *sample-1*

  eseguito per 1500 "generazioni";
- il *sample-2* è una matrice 400×400 contenente una massa di *sabbia* e una massa di *acqua*

[1] modello matematico e computazionale usato per descrivere l'evoluzione di sistemi complessi discreti attraverso semplici regole

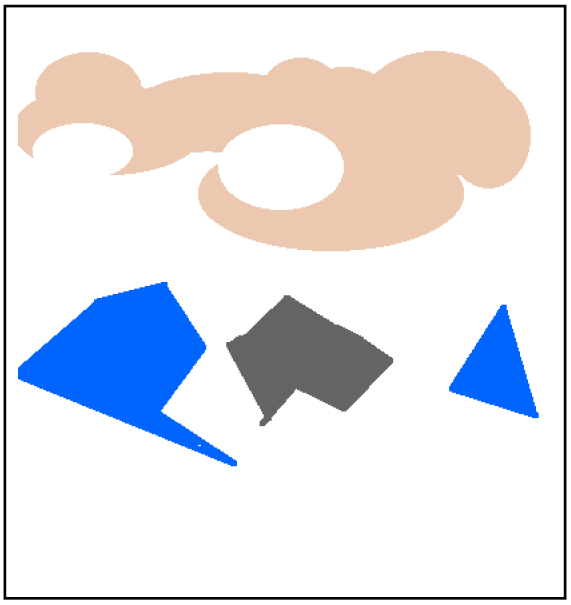

Figura 2: *sample-2*

eseguito per 1500 "generazioni";

- il *sample-3* e *sample-4* sono rispettivamente matrici 1920×1080 e 3840×2160 contenente masse di *sabbia* e *acqua* più numerose

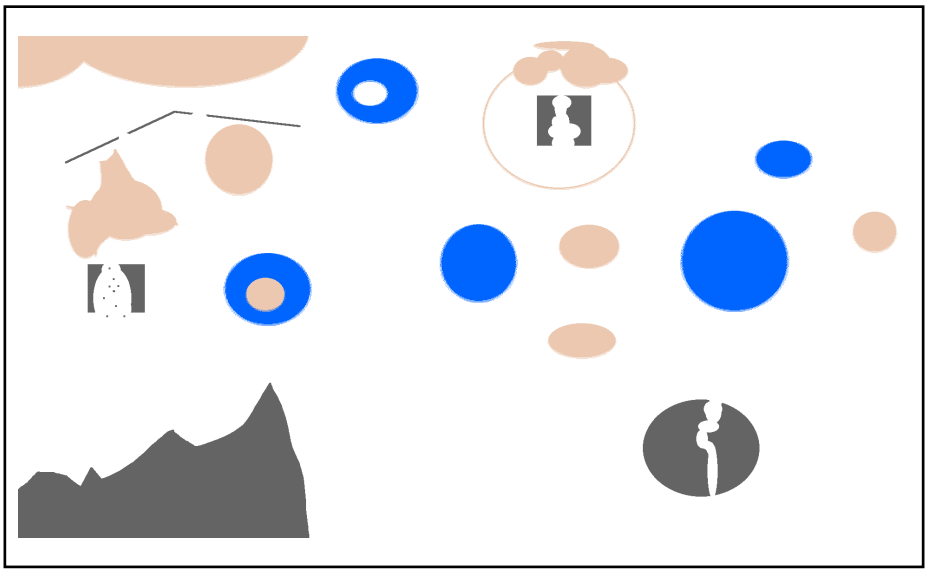

Figura 3: *sample-3* e *sample-4*

eseguito per 3000 iterazioni.

Si ipotizza che ogni matrice sia contenuta in un rettangolo delimitato da *muri* (per gestire eventuali accessi fuori dai limiti) in modo da conservare le particelle ad ogni iterazione.

1.1. Struttura del codice

Il programma realizzato è scritto in C e le operazioni che esegue ad ogni esecuzione sono:

- lettura dello stato iniziale dal file di input in formato `.sand`, particolare formato binario che codifica lo stato della matrice;
- esecuzione dell'algoritmo di simulazione per il numero di iterazioni richiesto e salvataggio di ogni stato intermedio in un file `.sand` indicato;
- eventuale **testing di corretta generazione** rispetto ad un file di riferimento `.sand` fornito;
- **eventuale generazione delle immagini** `.png` di ogni stato intermedio salvato nella cartella di output indicata.

1.1.1. Rappresentazione dei dati

Ogni stato dell'automa cellulare è rappresentato da una struct `Universe` così definita:

```C++
struct Universe {
    unsigned char *cells; // Vettore monodimensionale che rappresenta la matrice (row-major order)
    int width; // Larghezza della matrice
    int height; // Altezza della matrice
};
```

Al fine di semplificare la lettura dei dati, sono anche definite le seguenti costanti simboliche:

```C++
#define P_EMPTY 0
#define P_WATER 1
#define P_SAND 2
#define P_WALL 3
```

il cui valore rappresenta anche la "densità" della particella.

1.1.2. Formato `.sand`

Il formato `.sand` è un formato binario che codifica lo stato della matrice di simulazione. Dato che ogni particella è rappresentata da un `unsigned char` ma sono utilizzati solo 4 valori, ogni valore della matrice può essere memorizzato in soli 2 bit permettendo una maggiore compressione dei dati. Ogni file `.sand` è strutturato come segue:

- i primi 4 byte indicano il magic number `SAND` (in ASCII);
- i successivi 4 byte rappresentano la larghezza della matrice;
- i successivi 4 byte rappresentano l'altezza della matrice;
- i successivi 4 byte indicano il numero di frame salvati;
- i successivi width × height valori (ognuno rappresentato da 2 bit) rappresentano il vettore `cells` della struct `Universe` per ogni frame salvato, in ordine di generazione.

1.1.3. Algoritmo di simulazione

L'algoritmo di simulazione ha preso ispirazione dall'implementazione consultabile al seguente link [1], ma il risultato grafico non differisce particolarmente da altre implementazioni.
Particolarità di questa implementazione riguarda il fatto che ad ogni iterazione lo stato della matrice dipende solamente dallo stato precedente e non dallo stato in corso di aggiornamento.
Per farlo, utilizza come base il **pattern di Margolus**, studiato nella teoria degli automi cellulari e utile per ottenere la conservazione della massa e la reversibilità (non utile per il nostro caso).
Tale pattern prevede la divisione della matrice in blocchi 2×2 per cui si applicano le regole di aggiornamento che delineeranno l'evoluzione. Nelle iterazioni successive i blocchi saranno applicati con particolari offset, in modo da "applicare le regole" a tutte le celle della matrice.

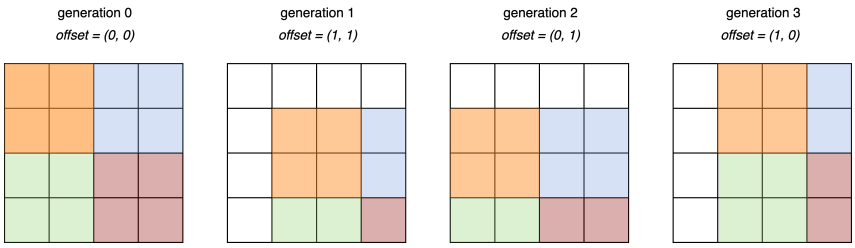

Figura 4: Pattern di Margolus

Le regole dedotte dall'implementazione [1] e utilizzate permettono di modificare ogni blocco in maniera "sequenziale" senza la necessità di tenere conto delle modifiche effettuate al blocco stesso, permettendo l'applicazione di ogni regola basandosi solamente sullo stato precedente. In particolare, le regole riguardano lo scambio di posizione delle particelle (in modo da garantire la conservazione della massa) sono le seguenti:

1. se una particella di *sabbia* si trova nella parte alta del blocco ed è "in caduta" (quindi si trovano sopra a particelle meno dense) è possibile (con una certa probabilità definita) che si muova orizzontalmente (scambiandosi di posizione con la particella ad essa adiacente nel blocco);
2. ipotizzando che la particella di *sabbia* non si sia mossa orizzontalmente, questa verifica che la particella sottostante sia meno densa; in questo caso è possibile (con una certa probabilità) che si muova verticalmente (scambiandosi di posizione con la particella sottostante); Nel caso la particella sottostante fosse più densa, la particella di *sabbia* potrebbe comunque spostarsi diagonalmente (se al suo fianco e nella sua destinazione fossero presenti particelle meno dense);
3. considerando ora la presenza di particelle d'*acqua*, queste si comportano in maniera simile alla *sabbia*, ma considerando il fatto che sia possibile (con una certa

probabilità) il movimento diagonale (sempre rispettando le regole di densità) ed il "galleggiamento" nonostante la possibilità di movimento verticale e/o diagonale
4. considerando invece il caso in cui delle particelle d'acqua si trovino sopra a particelle più dense (e non siano già cadute), queste potrebbero comunque spostarsi orizzontalmente con una certa probabilità;
5. sempre considerando il caso di particelle d'acqua, è necessario verificare l'eventuale presenza di particelle più dense al di sotto del blocco ed eventualmente (con una certa probabilità) spostarsi orizzontalmente (sempre rispettando le regole di densità);

È da notare che proprio per la struttura dell'algoritmo, qualsiasi **cambiamento ad un blocco composto da quattro celle identiche** non varierebbe la combinazione, rendendo inutile l'applicazione delle regole.
Per garantire il determinismo quando è necessario verificare la probabilità di un evento, è utilizzato un generatore di numeri pseudo-casuali basati sulla posizione della particella e sulla generazione (il numero del frame).
Tale logica, per essere applicata correttamente nel programma, deve essere implementata in una funzione che rispetta la firma

```C++
void next(Universe *in, Universe *out, int generation);
```

Non sono presenti vincoli sull'integrità dei dati in input, quindi è teoricamente possibile restituire come output in `out` esattamente il puntatore ad `in` con il contenuto modificato. Sarà sul tempo di esecuzione di tale funzione che saranno effettuate le misurazioni delle performance delle diverse ottimizzazioni.

1.2. Analisi del problema

Come già definito, il cuore delle diverse ottimizzazioni è rappresentato dalla funzione `next`, che deve essere eseguita per il numero di iterazioni richiesto. In ingresso è ricevuto un puntatore ad una struct `Universe`, contenente un vettore monodimensionale che rappresenta la matrice in *row-major order*. Come specificato nella definizione dell'algoritmo, non sono presenti dipendenze tra i dati di blocchi diversi ed è solamente necessario garantire la coerenza degli scambi all'interno dello stesso blocco. Ciò rende l'algoritmo **facilmente parallelizzabile** (*embarrassingly parallel*) e ottimizzabile per architetture vettoriali.

1.3. Implementazione base

L'implementazione sequenziale base (consultabile in `src/logic.c` e quindi `src/utility/utility-functions.c`) è stata realizzata in C ed è stata scritta in modo da essere il più leggibile possibile, utilizzando `if` e svolgendo ogni operazione in modo esplicito ed il meno complicato possibile. Nonostante questo, sono state adottate ottimizzazioni base per rendere l'implementazione più efficiente possibile senza compromettere la leggibilità del codice. Tra le ottimizzazioni adottate, si è evitato di processare blocchi completamente uguali e si sono utilizzate funzioni *inline* per evitare l'overhead di chiamata.

2 Ottimizzazioni SIMD

Di seguito sono descritte alcune ottimizzazioni che utilizzano istruzioni SIMD (*Single Instruction, Multiple Data*) per migliorare le prestazioni dell'implementazione base.
Per farlo è stata utilizzata la libreria Google Highway [2], che fornisce un'interfaccia portabile per l'utilizzo di istruzioni SIMD su diverse architetture.
L'idea alla base è che sia possibile elaborare più celle contemporaneamente caricando nei registri vettori che contengono i valori di più celle (e quindi di diversi blocchi).
Oltre ad adattare l'algoritmo per processare più blocchi contemporaneamente, è stato fatto un passo in più per rendere *branchless* l'algoritmo, ovvero evitare di utilizzare istruzioni di salto per gestire gli scambi. In alternativa sono state impiegate le seguenti istruzioni che permettono di eseguire lo scambio

```C++
template <class Descriptor, class Vector, class Mask>
HWY_INLINE void IfSwap(Descriptor d, Mask condition_mask, Vector &a, Vector &b)
{
    auto new_a = hn::IfThenElse(condition_mask, b, a);
    auto new_b = hn::IfThenElse(condition_mask, a, b);
    a = new_a;
    b = new_b;
}
```

Per determinare ogni condizione, si è utilizzata una combinazione di istruzioni logiche per calcolare le maschere di condizione, poi fornite alla funzione `IfSwap` per scambiare i corretti "componenti" del vettore caricato.
Per effettuare le trasformazioni, è stato necessario adattare ogni istruzione `if-else-if` ponendo attenzione a rispettare l'effettivo comportamento dell'algoritmo originale (ad esempio, per calcolare lo scambio che avviene nel ramo `else-if` è necessario negare la condizione del ramo `if` precedente e unirlo con la condizione del ramo `else-if`).
A causa delle dipendenze tra i dati all'interno di ogni "blocco", si ha che all'interno di ogni vettore sarebbe necessario effettuare confronti tra componenti adiacenti. Per farlo è quindi necessario effettuare delle permutazioni sui vettori caricati.
Sono state implementate diverse versioni dell'algoritmo, con leggere differenze.
Una versione "fallimentare"[2] è stata quella di utilizzare come istruzione di caricamento la funzione

```C++
hn::LoadInterleaved2(d, toprow_address + x, toplefts, toprights)
```

che permette di caricare in due vettori distinti tutti i dati in posizione pari (nel primo) e in posizione dispari (nel secondo) a partire da un indirizzo.
Tuttavia, nel nostro caso non si è rivelata una buona scelta, in quanto nei diversi *sample* presenterebbe performance peggiori rispetto alla versione base di riferimento. Il motivo

[2] di cui non sono presentati i risultati

di tale inefficienza è dovuto al fatto che per la struttura dei dati, sono presenti diversi blocchi composti da celle uguali che quindi non sono processati. Tale ottimizzazione è riportata anche in questa versione "fallimentare" ma non è sufficiente a compensare l'overhead introdotto dallo smistamento dei dati.
La soluzione[3] è stata quella di utilizzare per prima la funzione di caricamento `hn::LoadU` (che effettua il caricamento anche di dati non allineati, per rendere il codice compatibile con il programma preesistente[4]), per poi verificare se tutti i blocchi caricati sono composti da celle uguali tra loro[5], e in caso affermativo non processarli. In caso contrario, si procede con lo smistamento dei dati attraverso le funzioni `hn::ConcatEven` e `hn::ConcatOdd`.
Da notare che in entrambe le versioni ad ogni iterazione si processano esattamente il doppio di celle rispetto a quelle inseribili in un vettore (indicato dalla variabile `lanes` che è calcolata dalla libreria in base alla dimensione dei dati e all'architettura), in quanto i confronti vengono fatti tra i componenti adiacenti che devono essere separati, spostando i dati confrontabili in due vettori distinti.
L'ultima ottimizzazione implementata prevede l'utilizzo di istruzioni di *prefetch*[6] per caricare in anticipo i dati che saranno processati nei successivi passi dell'algoritmo.

2.1. Confronto delle prestazioni

Di seguito sono riportati i risultati delle esecuzioni dei diversi algoritmi sui vari *sample*. Tutti i test sono stati eseguiti su un computer MacBook Pro con processore Apple M2 e 8 GB di RAM e i risultati sono la media di tre esecuzioni.
Oltre alle versioni ottimizzate manualmente, sono stati inseriti anche i risultati delle versioni ottimizzate dal compilatore attraverso l'utilizzo del flag `-O3`.

[3] consultabile nel file `src/simd/simd-manual-interleave.cpp`

[4] scelta progettuale per evitare di rendere dipendente da Google Highway il codice principale

[5] si intende che le celle di ogni blocco devono essere uguali tra loro all'interno del blocco, non è necessario che anche i blocchi siano uguali tra loro

[6] consultabile nel file `src/simd/simd-manual-interleave-prefetch.cpp`

Versione	**Numero cicli**	**Speedup**
Base	17 633 412	$\times$ 1
Base con ottimizzazione `-O3`	4 935 159	$\times$ 3.57
Ottimizzazione SIMD senza prefetch	2 511 367	$\times$ 7.02
Ottimizzazione SIMD con prefetch	2 520 964	$\times$ 6.99

Tabella 1: Risultati ottimizzazione SIMD del sample 1 (400×400) per 1500 frame

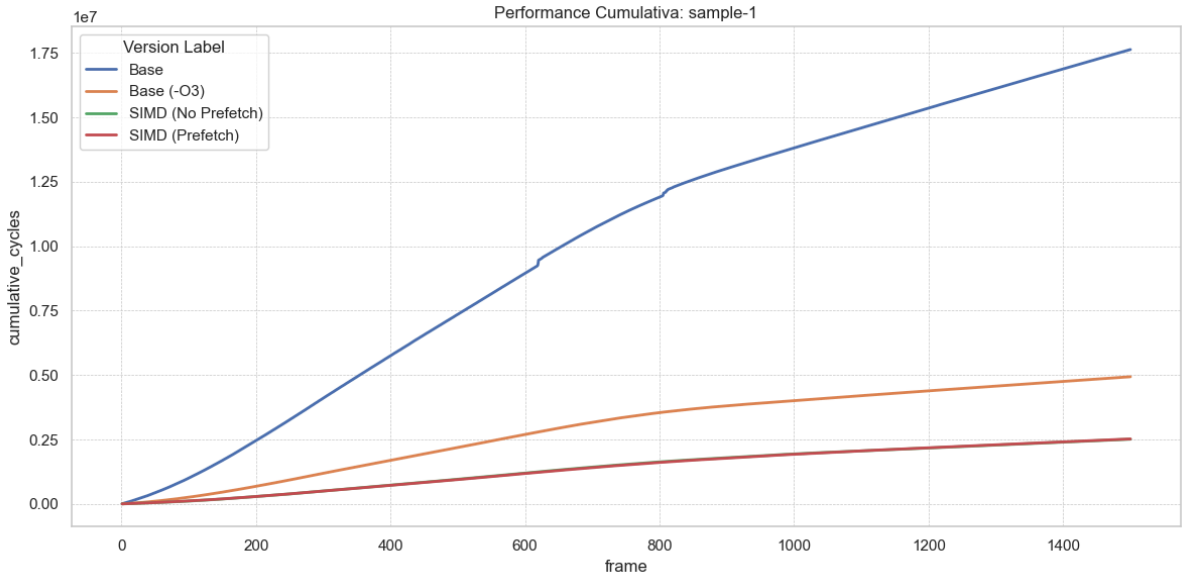

Figura 5: Grafico dei risultati per le ottimizzazioni SIMD del sample 1

Versione	**Numero cicli**	**Speedup**
Base	18 760 698	$\times$ 1
Base con ottimizzazione `-O3`	5 640 478	$\times$ 3.33
Ottimizzazione SIMD senza prefetch	2 718 206	$\times$ 6.9
Ottimizzazione SIMD con prefetch	2 858 030	$\times$ 6.56

Tabella 2: Risultati ottimizzazione SIMD del sample 2 (400×400) per 1500 frame

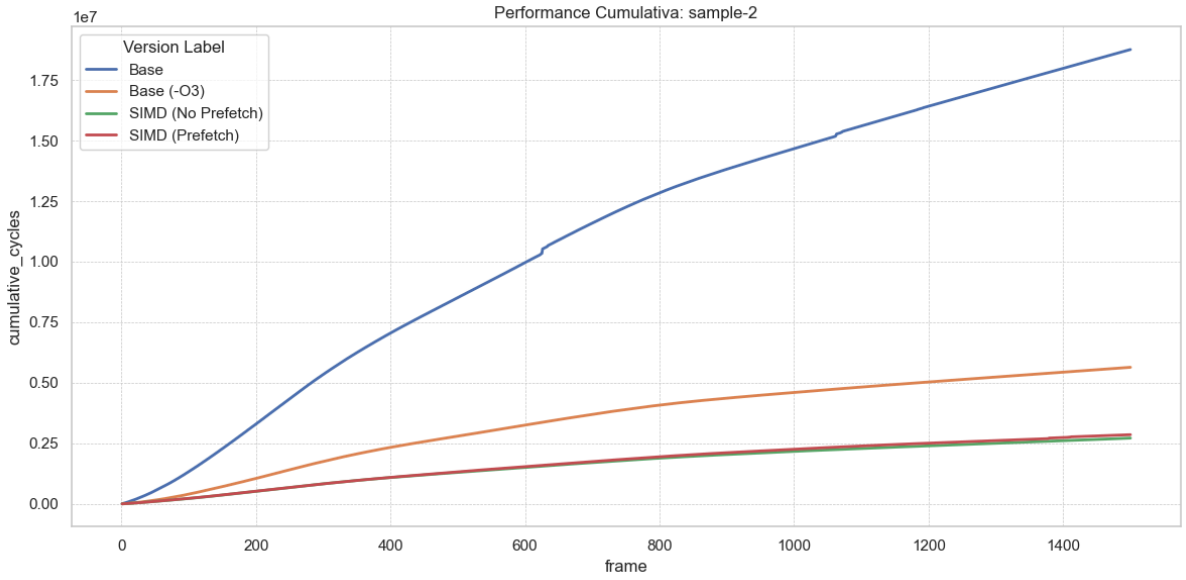

Figura 6: Grafico dei risultati per le ottimizzazioni SIMD del sample 2

Versione	**Numero cicli**	**Speedup**
Base	392 756 058	$\times$ 1
Base con ottimizzazione `-O3`	111 089 334	$\times$ 3.53
Ottimizzazione SIMD senza prefetch	53 052 551	$\times$ 7.4
Ottimizzazione SIMD con prefetch	53 404 369	$\times$ 7.35

Tabella 3: Risultati ottimizzazione SIMD del sample 3 (1920×1080) per 3000 frame

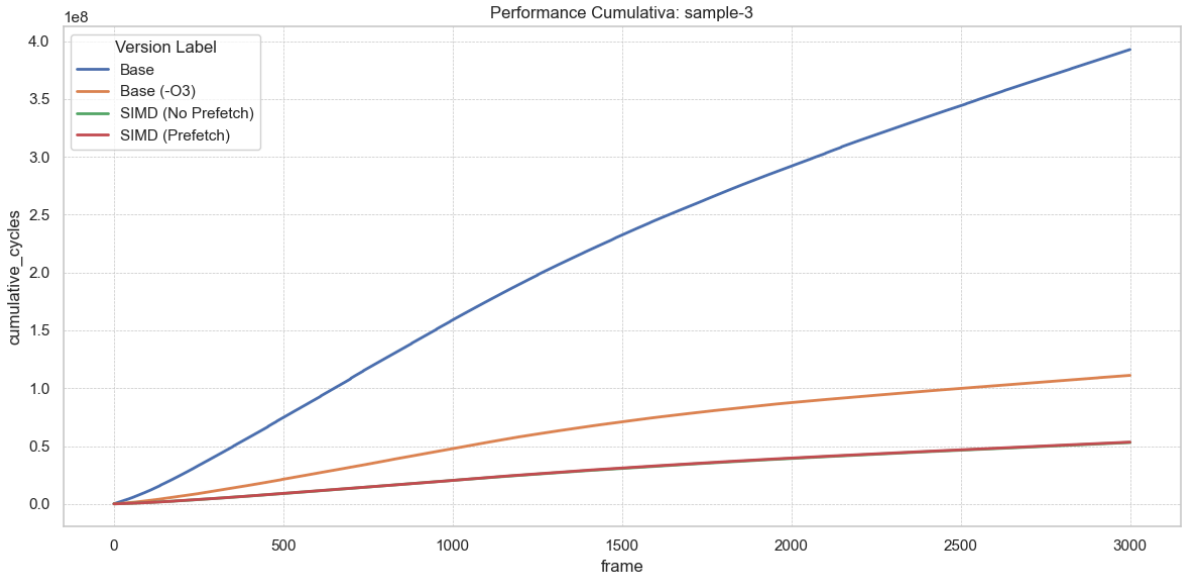

Figura 7: Grafico dei risultati per le ottimizzazioni SIMD del sample 3

Versione	Numero cicli	Speedup
Base	507 135 947	× 1
Base con ottimizzazione `-O3`	510 808 022	× 0.99
Ottimizzazione SIMD senza prefetch	206 792 052	× 2.45
Ottimizzazione SIMD con prefetch	209 215 278	× 2.42

Tabella 4: Risultati ottimizzazione SIMD del sample 4 (3840 × 2160) per 3000 frame

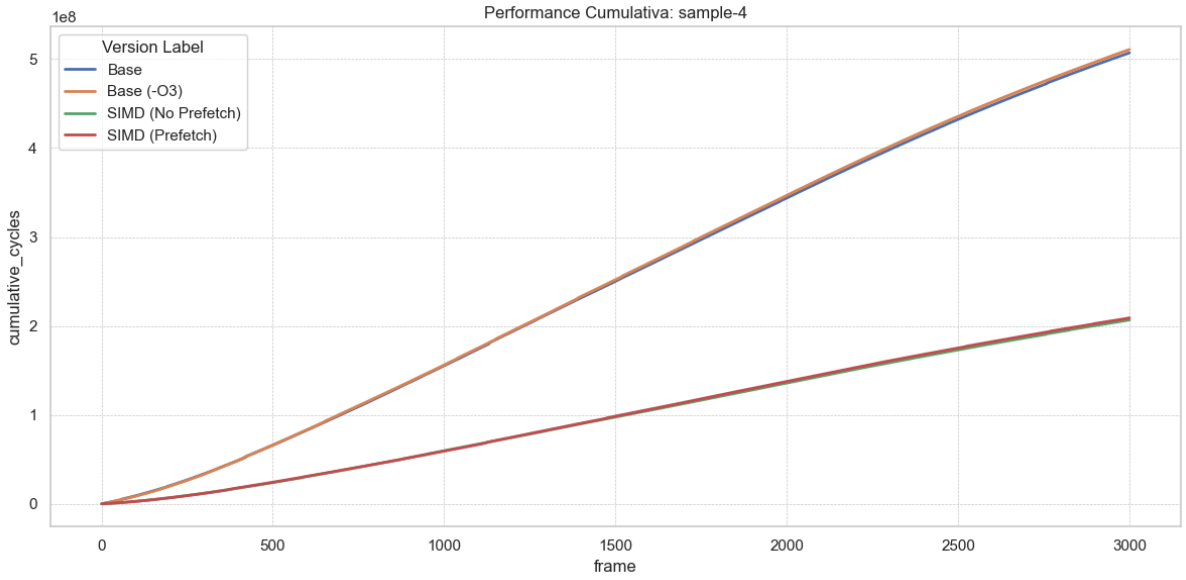

Figura 8: Grafico dei risultati per le ottimizzazioni SIMD del sample 4

Come è possibile notare dai risultati, le ottimizzazioni SIMD implementate permettono di ottenere un miglioramento significativo delle prestazioni rispetto alla versione base, con uno speedup di circa 7 volte (leggermente migliore per il *sample 3*, dovuto probabilmente al maggior numero di frame coinvolti, e di molto peggiore per il *sample 4*), anche superiore rispetto all'ottimizzazione fatta automaticamente dal compilatore. I risultati sono coerenti per tutti i *sample* ed è possibile notare che l'ottimizzazione con prefetch peggiora generalmente le prestazioni (probabilmente dovuto all'overhead introdotto dalle istruzioni che non compensa il vantaggio di avere i dati già caricati automaticamente).

Il miglioramento è dovuto alla combinazione del vantaggio ottenuto dalla versione *branchless* e dalla maggiore quantità di dati processati contemporaneamente offerta da SIMD.

3 Ottimizzazione CUDA

Comne visto nella parte di analisi del problema, l'algoritmo risultera essere particolarmente parallelizzabile nell'esecuzione di ogni frame, in quanto i dati iniziali sono presenti in una matrice. L'unica dipendenza fra i dati si trova all'interno di ogni blocco 2×2, in cui gli scambi devono essere consistenti.

3.1. Implementazione naive

La prima implementazione è stata quella *naive*, in cui si è cercato di mantenere una certa leggibilità del codice ed equivalenza rispetto alla versione sequenziale.
Le variazioni rispetto alla versione sequenziale si trovano ovviamente nella parte di "lancio" del kernel (cambia la funzione `next`).
Per questa prima versione, a seguito dell'analisi di diverse alternative, è stata scelta una dimensione di blocchi di 32×32.
Dato che questa prima versione è analoga alla versione sequenziale, ogni kernel dovrà gestire blocchi di 2×2 elementi della matrice iniziale: da questo deriva il calcolo della dimensione della griglia:

```C++
dim3 grid((u_in->width / 2 + block.x - 1) / block.x,
          (u_in->height /2 + block.y - 1) / block.y);
```

Inoltre, per rispettare totalmente la logica di gestione della memoria fatta del caso sequenziale, ad ogni invocazione della funzione è copiata in memoria del *device* la matrice di input e copiata in memoria dell'host la matrice di output.

3.2. Implementazione ottimizzata (*optimized*)

Questa versione mantiene la struttura della versione precedentemente descritta. Si sono effettuate piccole ottimizzazioni a seguito del profiling. Tra le modifiche effettuate:

- si sono eliminate le chiamate a funzione, in modo da diminuire l'overhead;
- sono state diminuite le strutture di selezione per aumentare la *branch prediction*;
- le istruzioni sono state svolte su variabili valore: nella versione base si effettuava continuamente la deferenziazione dei puntatori per ottenere il valore. In questa ottimizzazione sono state usate variabili locali al kernel per la gestione delle celle. Leggendo all'inizio del kernel i valori, si evitano i continui accessi alla matrice `in`.

3.3. Versioni *branchless*

A seguito delle precedenti implementazioni, si è cercato di migliorare ulteriormente il kernel in modo da renderlo ancora più efficiente.

Osservando la versione naive, è stata evidente la presenza di molte istruzioni di selezione `if`, che aumentano la warp divergence.
La soluzione adottata nelle successive versioni è stata quella di eseguire in ogni caso le istruzioni (attraverso funzioni inline per evitare l'overhead) per effettuare lo scambio dei valori delle variabili solo ad una condizione, evitando strutture condizionali in favore di operazioni bitwise che effettuano lo scambio solo se la condizione risulta vera

```C++
__device__ inline void ifSwap(bool condition, unsigned char *a,
unsigned char *b){
    unsigned char mask = -condition; // 0xFF se true, 0x00 se false
    unsigned char temp = (*a ^ *b) & mask;
    *a = *a ^ temp;
    *b = *b ^ temp;
}
```

3.3.1. Ottimizzazione block branchless

Questa ottimizzazione riporta esattamente la stessa logica della versione precedente, ma con l'utilizzo di istruzioni bitwise per effettuare lo scambio dei valori invece di strutture condizionali.

3.3.2. Branchless single thread

Per questa versione si è utilizzato un approccio più *GPU-friendly*, per cui ogni thread avesse il compito di aggiornare la singola cella ad esso assegnata.
Tuttavia ciò ha portato ad un aumento dei calcoli necessari a mantenere la consistenza all'interno dei blocchi 2×2.
Oltre a questo, in questa ottimizzazione non è possibile aggiornare la grigia 2×2 interamente, portando alla decisione di creare la funzione (sempre *inline* per evitare overhead) che restituisce in ogni caso il valore futuro assunto dalla cella. Questa "discriminazione" è fatta attraverso il seguente codice che ricorda la logica di una lookup table hardware

```C++
int mask_x = -x;
int not_mask_x = ~mask_x;
int mask_y = -y;
int not_mask_y = ~mask_y;
return (topleft & not_mask_x & not_mask_y) |
       (topright & mask_x & not_mask_y) |
       (bottomleft & not_mask_x & mask_y) |
       (bottomright & mask_x & mask_y);
```

Oltre a ciò, per questa versione è stato ritenuto utile sfruttare la *shared memory* messa a disposizione in quanto si adattava particolarmente alla natura del problema.

```C++
 __shared__ unsigned char s_tile[TILE_DIM_Y][TILE_DIM_X];
```

Il vantaggio ottenuto da tale implementazione riguarda il fatto che ogni thread per calcolare il suo stato futuro deve accedere a dati che si trovano nel suo intorno, in modo da diminuire gli accessi in memoria globale.
Questo approccio porta ovviamente ad un aumento di thread necessari, esattamente il quadruplo.

3.4. Confronto delle prestazioni

Dopo la fase di implementazione si è proceduto con la valutazione delle prestazioni ed il profiling tramite NVIDIA Nsight Compute.
Tutti i test sono stati effettuati su un elaboratore con le seguenti specifiche:

Sistema operativo	Kali Linux
Processore host	12th Gen Intel(R) Core(TM) i7-1255U
Host Memory	16 GB
CUDA version	12.4
NVIDIA Device	NVIDIA GeForce MX550
Device Memory	2 GB
Device Architecture	Turing
Device Compute Capability	7.5

Tabella 5: Specifiche elaboratore su cui sono stati effettuati test

A seguito di una prima analisi delle prestazioni si è notato che le prestazioni delle varie versioni non variavano di tanto fra l'una e l'altra, portando alla scelta di rendere le versioni branchless (considerate le più promettenti) *stateful*. In questo modo, la matrice è caricata dall'host solo la prima volta e utilizzare quella aggiornata direttamente dallo stato precedente.
Per essere consistenti con il resto del programma e mantenendo il fatto che ad ogni iterazione il kernel deve analizzare solo il singolo frame, ad ogni iterazione è necessario copiare la matrice risultante sull'host.

Versione	**Numero cicli**	**Speedup**
Base (sequenziale)	1 312 600 611	$\times$ 1
Naive	987 677 573	$\times$ 1.33
Ottimizzata	906 587 877	$\times$ 1.45
Branchless a blocchi	397 147 967	$\times$ 3.31
Branchless single thread	493 391 021	$\times$ 2.66

Tabella 6: Risultati ottimizzazione CUDA del sample 1 (400×400) per 1500 frame

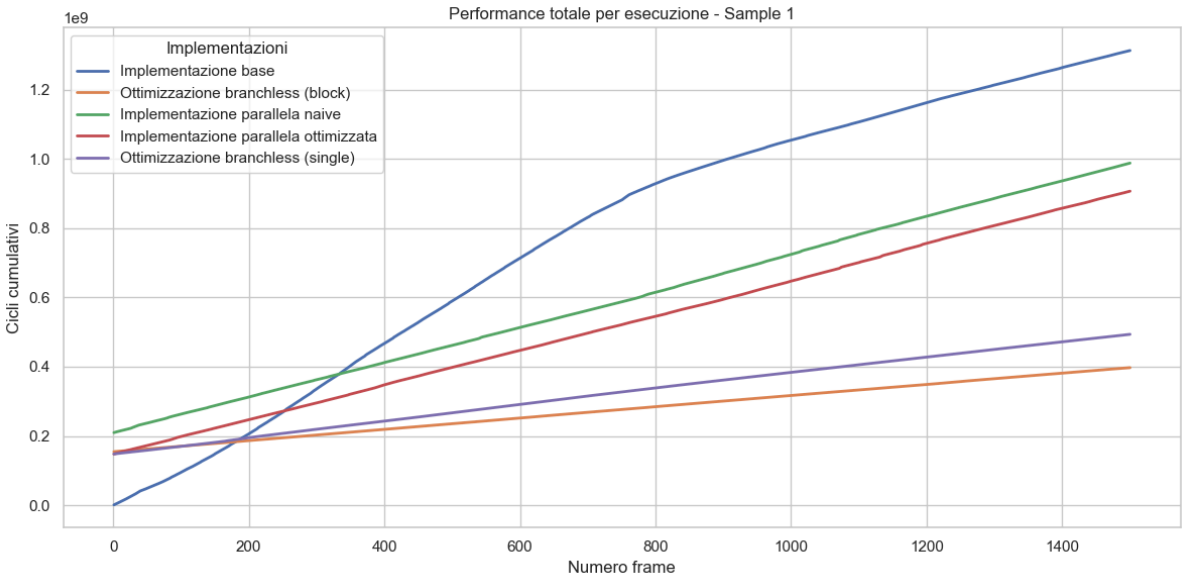

Figura 9: Grafico dei risultati per le ottimizzazioni CUDA del sample 1

Versione	Numero cicli	Speedup
Base (sequenziale)	1 446 900 357	$\times$ 1
Naive	996 944 612	$\times$ 1.45
Ottimizzata	970 225 327	$\times$ 1.5
Branchless a blocchi	401 415 193	$\times$ 3.6
Branchless single thread	503 544 295	$\times$ 2.87

Tabella 7: Risultati ottimizzazione CUDA del sample 2 (400×400) per 1500 frame

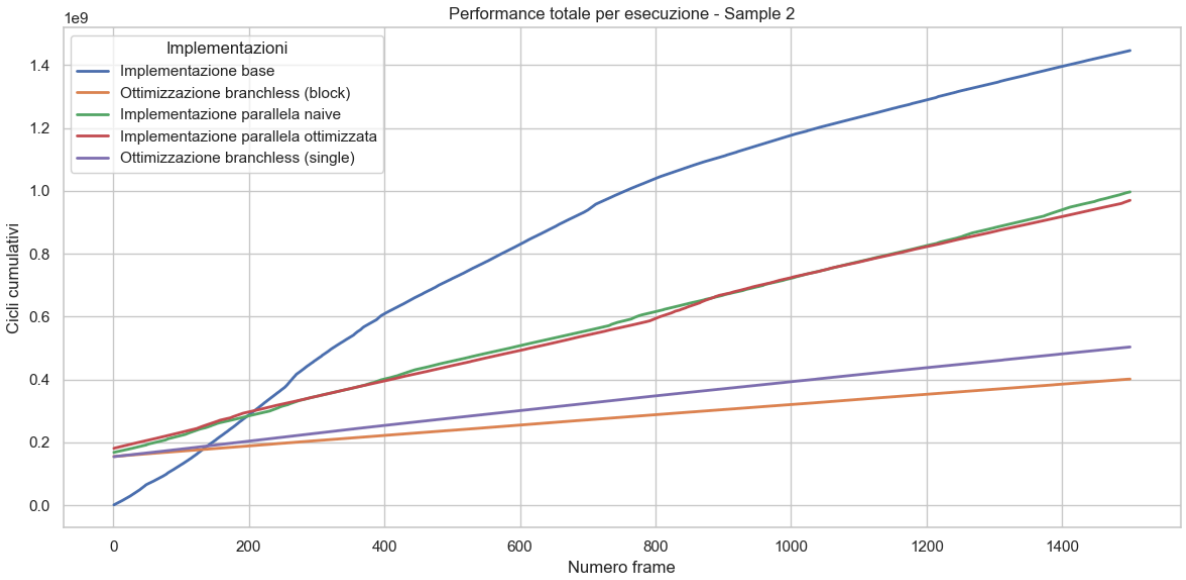

Figura 10: Grafico dei risultati per le ottimizzazioni CUDA del sample 2

Versione	Numero cicli	Speedup
Base (sequenziale)	31 522 709 229	$\times$ 1
Naive	13 340 281 229	$\times$ 2.36
Ottimizzata	13 100 873 649	$\times$ 2.4
Branchless a blocchi	4 833 328 935	$\times$ 6.5
Branchless single thread	7 440 696 748	$\times$ 4.23

Tabella 8: Risultati ottimizzazione CUDA del sample 3 (1920×1080) per 3000 frame

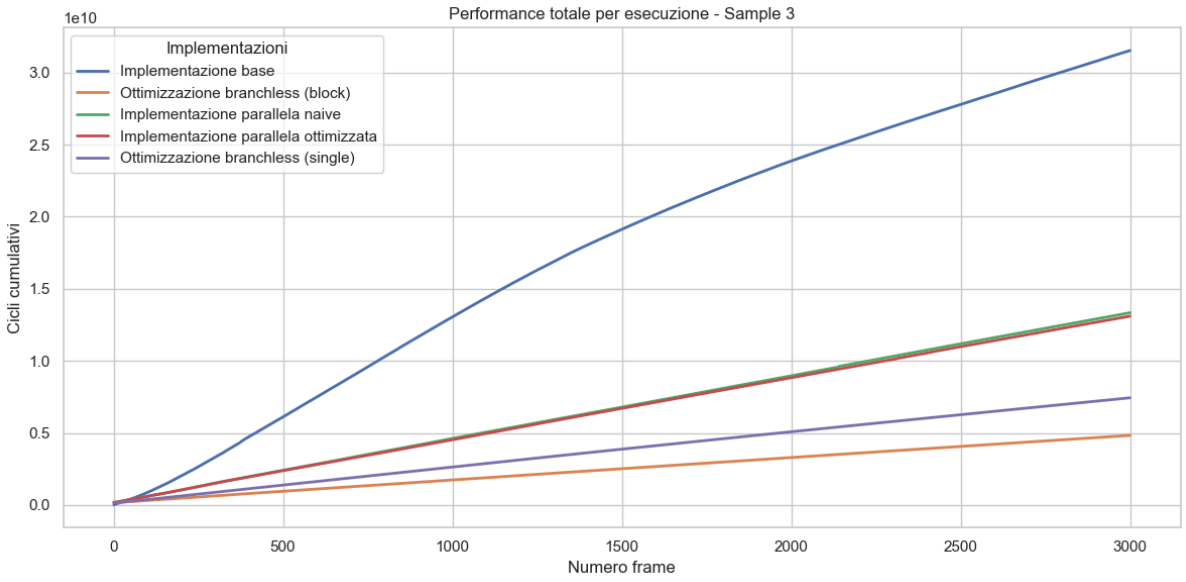

Figura 11: Grafico dei risultati per le ottimizzazioni CUDA del sample 3

Versione	Numero cicli	Speedup
Base (sequenziale)	139 511 260 260	× 1
Naive	42 068 946 161	× 3.32
Ottimizzata	41 210 448 728	× 3.39
Branchless a blocchi	16 857 574 178	× 8.28
Branchless single thread	27 644 583 721	× 5.05

Tabella 9: Risultati ottimizzazione CUDA del sample 4 (3840 × 2160) per 3000 frame

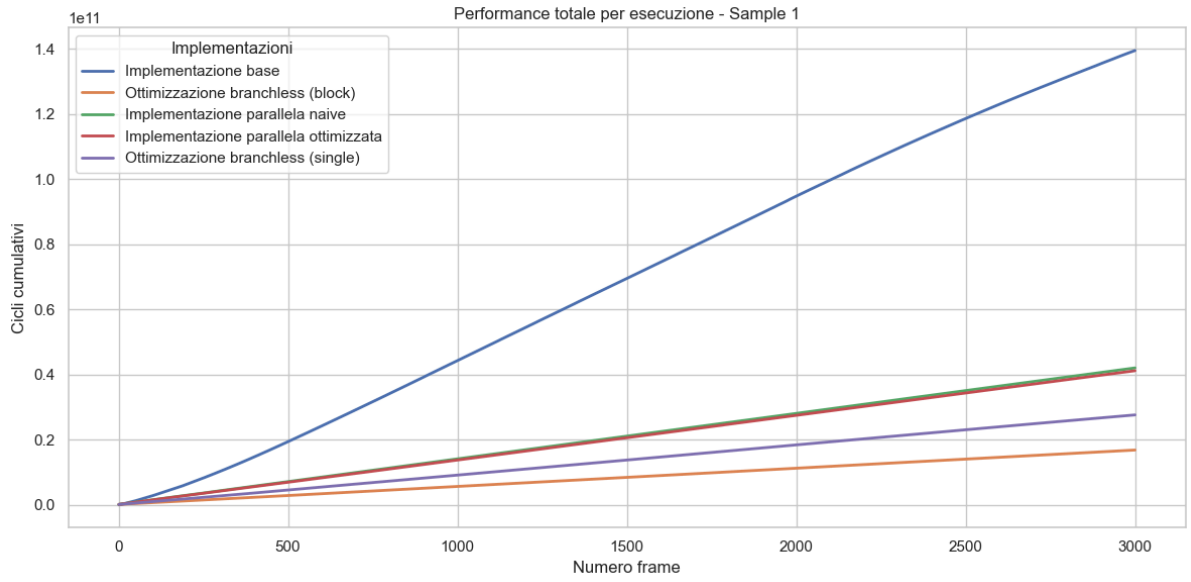

Figura 12: Grafico dei risultati per le ottimizzazioni CUDA del sample 4

3.4.1. Profiling

Il profiling è stato fatto tramite NVIDIA Nsight Compute.
Prima di tutto, è importante notare che il kernel sviluppato svolge tante operazioni bitwise, shift e calcolo di indirizzi. Le uniche operazioni floating point effettuate riguardano l'operazione floating point per la generazione di numeri random, che tuttavia non influisce sull'utilizzo della bandwidth e non è modificabile per tener fede all'algoritmo originale. Per questo motivo, si è deciso di non effettuare l'analisi del *roofline model*.

Altra osservazione importante riguarda il fatto l'algoritmo è fortemente memory bound e che le ultime ottimizzazioni cercano di ovviare a tale problema utilizzando la shared memory.
Attraverso la funzione `ifSwap` (descritta in precedenza) si svolge lo scambio dei valori solo se la condizione è vera, permettendo di diminuire la *warp divergence* e a rendere coerente tra i thread la compute capability, riuscendo a mantenere l'*occupancy* in un range ottimale.

3.4.2. Versione naive e ottimizzata

Tale valore è soddisfacente già nella versione naive (intorno al 75%): nonostante ciò all'aumentare dei dati il tempo di esecuzione risulta essere peggiore rispetto a quello delle altre versioni ottimizzate in parallelo, seppur più veloce rispetto alla versione sequenziale. Contribuisce a ciò anche l'hit rate delle memorie cache, pari a 54% per la cache L1 e 61% per la cache L2.
La versione leggermente ottimizzata mostra prestazioni lievemente migliori ma comunque molto simili alla versione naive.

3.4.3. Versioni branchless

Dati più interessanti sono quelli delle implementazioni branchless.
L'implementazione del kernel *block branchless* risulta essere la più veloce, ottenendo uno speedup di circa 6 volte e mezzo. Presenta un'*occupancy* leggermente più alta rispetto alla versione naive (pari circa all'80%), valori di *cache hit* maggiori pari al 58% per L1 e picchi del 74% (nei diversi frame).
Considerando una media di 25 thread attivi sui 32 massimi per warp, si è deciso tenendo in considerazione i risultati del profiling, di diminuire la dimensione dei blocchi da 32×32 a 16×16 per mantenere un numero di thread attivi più alto possibile. Fisiologicamente diminuisce anche l'*occupancy* arrivando ad un valore pari a $\frac{77}{78}\%$ ma portando anche ad un miglioramento dei tempi di esecuzione. Variando la dimensione del blocco, si ottiene anche un miglioramento del memory throughput, che si stabilizza intorno ai 50 Gbyte/sec rispetto ai $\frac{30}{40}$ Gbyte/sec precedenti..
L'implementazione del kernel *single branchless* (in cui ogni thread aggiorna una singola cella) prevedeva inizialmente l'accesso alla memoria globale, portnado ad un tempo di esecuzione spaventosamente alto (caratterizzato da un'occupancy elevata pari ad oltre il 90% e ad un aumento della cache hit). Utilizzando la shared memory ha portato ad un miglioramento significativo, anche se comunque inferiore rispetto alla versione *block branchless*. Anche in questo caso, si è deciso di diminuire la dimensione dei blocchi da 32×32 a 16×16 ottenendo miglioramenti simili.

4 Conclusioni

In conclusione si può affermare che le ottimizzazioni effettuate hanno permesso di ottenere un miglioramento significativo delle prestazioni utilizzando sia ottimizzazioni vettoriali, sia attraverso operazioni di calcolo parallelo.
In particolare l'ottimizzazione SIMD ha permesso di ottenere un miglioramento di circa 7 volte rispetto alla versione base, anche superiore rispetto all'ottimizzazione fatta automaticamente dal compilatore, con risultati coerenti per tutti i *sample*. Le ottimizzazioni di calcolo parallelo hanno permesso di ottenere un miglioramento inferiore per i sample più piccoli (a causa dell'overhead introdotto dal caricamento dei dati sul device) che diventa più significativo per il sample di dimensione maggiore (1920×1080 e soprattutto per la versione 3840×2160) per un maggiore numero di frame, con uno speedup di oltre 8 volte per la versione *block branchless*. Il principale collo di bottiglia individuato (nella versione parallela) risulta essere legato all'accesso in memoria dei dati che causa un overhead significativo.

5 Bibliografia

[1] GelaMiSalami, «GPU Falling Sand CA». [Online]. Disponibile su: https://gelamisalami.github.io/GPU-Falling-Sand-CA/

[2] J. Wassenberg e G. LLC, «Highway: Performance-portable, length-agnostic SIMD with runtime dispatch». [Online]. Disponibile su: https://github.com/google/highway