Sistemi Digitali

Ottimizzazione di sistemi di particelle in CUDA

Implementazione e analisi delle performance di una simulazione di un fuoco

Federica Di Giaimo

Il Problema

Simulazione di sistemi di particelle

- Tecnica fondamentale nella computer grafica.
- Simulazione in tempo reale con un alto numero di particelle è computazionalmente costoso per una CPU tradizionale.

Data Parallelism

- Le particelle rappresentano un enorme set di dati.
- Regole fisiche o procedurali applicate ad ogni particella per ogni frame.

Il Problema

Obiettivo del progetto

- Sfruttare l'architettura parallela delle GPU tramite CUDA per accelerare una simulazione di particelle per il fuoco.
- Parallelizzazione e ottimizzazione in modo efficiente permettendo un alto grado di realismo e interattività in tempo reale.

Applicazioni Pratiche

- Videogiochi.
- Film e VFX.
- Visualizzazioni scientifiche.
- Realtà aumentata.
- Simulazioni di fluidodinamica.

Analisi del problema

Natura dei Dati

- Array di N particelle, ognuna con stato.
- Calcoli in virgola mobile per aggiornarne lo stato a ogni frame.

Dipendenze dei Dati

Modello base

- Sistema con particelle indipendenti.
- Lo stato di una particella dipendono solo dal suo stato precedente.

Modello avanzato

- Sistema con particelle interagenti.
- Lo stato di una particella dipende anche dallo stato delle sue vicine.

Parallelizzazione

Modello base

- Problema "Embarrassingly Parallel"
- Parallelismo Massivo dei Dati

Modello avanzato

- I thread ora devono comunicare e condividere dati tra loro.
- Decomposizione del dominio con comunicazione locale

Metodologia e Baseline

Approccio incrementale e iterativo

1) Baseline CPU (1_naive.cu)

2) Parallelo naive (1_naive.cu)

3) Ottimizzazione computazionale e dell'occupancy (2_optimized_warp_divergence.cu)

4) AoS → SoA (2_optimized_warp_divergence.cu)

5)Implementazione shared memory (3_sharedmem.cu)

6) Interazione tra thread con shmem (4_interaction_particles)

CC 8.6 (NVIDIA 3060 RTX)

Physical Limit of GPU (8.6):

Property	Limit
Threads per Warp	32
Max Warps per Multiprocessor	48
Max Thread Blocks per Multiprocessor	16
Max Threads per Multiprocessor	1536
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	8192
Max Shared Memory per Block	8192
Register Allocation Unit Size	256
Register Allocation Granularity	warp
Shared Memory Allocation Unit Size	128
Warp Allocation Granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	1024

File: 1_naive.cu

Struttura del kernel naive parallelo

```
__global__ void updateParticlesKernel(Particle* particles, ParticleVertex* vbo_ptr,
    int numParticles, float dt, curandState* randStates, float time, int gridWidth)
{
    int global_x = blockIdx.x * blockDim.x + threadIdx.x;
    int global_y = blockIdx.y * blockDim.y + threadIdx.y;

    int idx = global_y * gridWidth + global_x;

    curandState localState = randStates[idx];
    Particle p = particles[idx];

    if (p.lifetime > 0.0f)
    {
        // Logica di aggiornamento (velocità, posizione)
        p.lifetime -= dt;
    }
    else
    {
        // Logica di rinascita con curand_uniform...

    particles[idx] = p;
    vbo_ptr[idx].x = p.posX;
    vbo_ptr[idx].y = p.posY;
    vbo_ptr[idx].z = 0.0f;
    randStates[idx] = localState;
}
```

Mapping 1 a 1 tra thread e particelle

Tutti gli accessi ai dati avvengono direttamente da e verso la memoria globale.

La logica di calcolo è completamente indipendente

! Warp divergence

- Tempo di esecuzione CPU (Baseline): ~30000 µs (o 30 ms)
- Tempo di esecuzione GPU (Naïve Parallelo): ~2000 µs (o 2.0 ms)

$$\text{Speedup} = \frac{\text{Tempo CPU}}{\text{Tempo GPU}} = \frac{30000\,\mu s}{2000\,\mu s} = 15$$

Struttura della baseline sequenziale

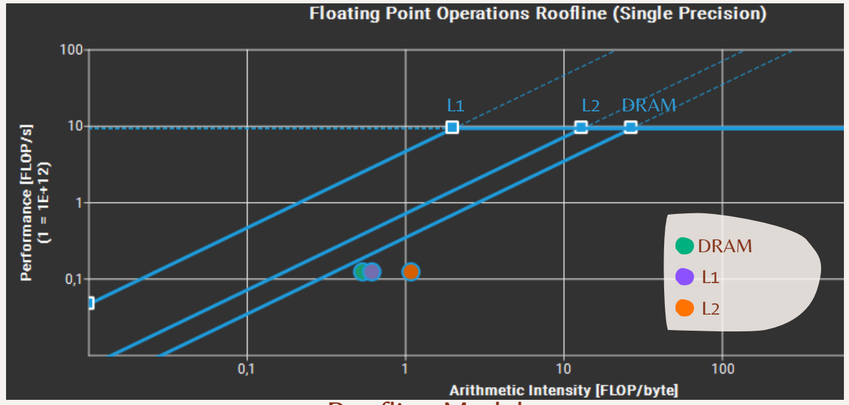
```
#ifdef SEQUENTIAL_BASELINE
void updateParticlesCPU(Particle* particles, int numParticles, float dt, float time)
{

    for (int idx = 0; idx < numParticles; idx++) {
        Particle p = particles[idx];

        //Particella viva
        if (p.lifetime > 0.0f) {
            p.velX -= p.posX * 1.2f * dt;
```

Un singolo thread esegue tutto il lavoro

Analisi della Baseline

Bottleneck: Memory bound

- Compute (SM) Throughput [%]: 6,40%
- Memory Throughput [%]: 75,49%



Roofline Model

Occupancy Calculator

- Occupancy teorica: 100%
- Occupancy Limiters: Max Warps per SM

Source Statistics

- Branch Efficiency: 90%

Fuorviante. Il compilatore applica numerose ottimizzazioni

Scheduler Statistics

Max Active Warps Per Scheduler in Ampere: 12

- Active Warps Per Scheduler: 11,35
- Eligible Warps Per Scheduler: 0,12
- No Eligible (warp) [%]: 96,12%
- Issue Slot Utilization [%]: 23.84%

Memory Workload Analysis

- Memory Throughput [Gbyte/s]: 221,25 GB/s
- Max Bandwidth [%]: 76,16%

Fuorviante. Dovuto ad uso intensivo della curandState.

- L1/TEX Hit Rate [%]: 78,26%
- L2 Hit Rate [%]: 90,75%

- Mem Busy [%]: 69,98%

Latency-bound

Warp State Statistics

- Avg. cycles stalled waiting for LG memory operations: 215.2 cycles
- Percentage of total cycles stalled for LG memory: 73,5%

Stalli dovuti alle dipendenze dalla global memory.

Ottimizzazione: Shared Memory

Studio della configurazione e dell'Occupancy

Configurazione 2D →
- CON: Complessità di Indicizzazione
- CON: Rischio per la coalescenza della memoria

Configurazione 1D con Massimi Thread per Blocco (1024) →
- PRO: Massimizzare parralellismo all'interno di ogni thread block
- CON: Consumo elevato registri SM
  → un solo thread block attivo su SM

Configurazione 1D con 512 thread per blocco →
- PRO: 3 thread block attivi su SM contemporaneamente
  → massimizza il numero di warp attivi per SM

NVIDIA Nsight Compute	(1024,1,1)	(512,1,1)
Active Warps Per Scheduler	6.7	10.3
Occupancy teorica	66.7%	100%

Size
(2048, 1, 1)x(512, 1, 1)

Riorganizzazione della memoria

File:
2_optimized_warp_divergence.cu

AoS

```
// Stato particella
typedef struct {
    float posX, posY;
    float velX, velY;
    float lifetime;
} Particle;
```

- Per leggere una singola proprietà, ad esempio posX, ogni thread di una warp carica l'intera struct

SoA

```
struct ParticlesSoA {
    float* posX; float* posY;
    float* velX; float* velY;
    float* lifetime;
    float* turbulence_flag;
    unsigned int* rand_state;
};
```

- I dati sono stati riorganizzati in una struct di puntatori, dove ogni campo punta a un array contiguo.
- Quando una warp legge ad esempio posX, c'è una sola transazione di memoria
- Lettura perfettamente coalesced.

NVIDIA Nsight Compute	Kernel AoS	Kernel SoA
Latenza (Cicli/Istruzione)	137	41.2
% Tempo inattivo GPU (No Eligible)	96.7%	83.5%
Stalli di memoria (Cicli)	77	30.8

Ottimizzazione computazionale

File: fast_math.cuh

- Libreria cuRAND sostituita con generatore Xorshift
  - Riduzione drastica dell traffico legato alla generazione di numeri casuali
  - Diminuzione latenza logica di respawn

- Funzioni sin e cos rimpiazzate da una Lookup Table implementata in __constant__ memory
  - Operazioni a latenza elevata trasformate in letture di memoria a bassissima latenza

```
//      OTTIMIZZAZIONE COMPUTAZIONALE (PRNG Leggero)
//
// Sostituzione di cuRAND con un generatore Xorshift molto più leggero.
// Riduce il traffico di memoria e la latenza di calcolo.
__device__ unsigned int xorshift32(unsigned int& state) {
    state ^= state << 13;
    state ^= state >> 17;
    state ^= state << 5;
    return state;
}

__device__ float random_float(unsigned int& state) {
    return (float)xorshift32(state) / 4294967295.0f;
}

//      OTTIMIZZAZIONE COMPUTAZIONALE (Lookup Table)
// Sostituzione delle funzioni sin() e cos() con una lettura
// dalla __constant__ memory.
__device__ __forceinline__ float fast_sin(float x) {
    // ... (corpo della funzione) ...
    return fmaf(frac, s1 - s0, s0);
}

__device__ __forceinline__ float fast_cos(float x) {
    return fast_sin(x + 1.57079632679f);
}

void setupSinLut() {
    // ... (corpo della funzione) ...
}
```

Ottimizzazione della latenza

File: 2_optimized_warp_divergence.cu

```cpp
#define SUB_STEPS 8

__global__ void fireKernel_SoA_FINAL(
    float* p_posX, float* p_posY,
    float* p_velX, float* p_velY,
    float* p_lifetime, unsigned int* p_rand_state,
    int numParticles,
    float dt,
    float time)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= numParticles) return;

    // ... (lettura dei dati in variabili locali/registri) ...

    //            OTTIMIZZAZIONE DELLA LATENZA (Sub-Steps)
    // Eseguiamo la logica di simulazione più volte dopo una singola lettura
    // dalla memoria globale. Questo aumenta l'intensità aritmetica e permette
    // alla GPU di nascondere la latenza di memoria eseguendo calcoli utili.
    const float sub_dt = dt / (float)SUB_STEPS;
#pragma unroll
    for (int i = 0; i < SUB_STEPS; ++i)
    {
        // ... (tutta la tua logica di fisica e respawn qui dentro) ...
    }
    // ... (scrittura dei dati una sola volta alla fine) ...
}
// ... (kernel updateVBOKernel_SoA) ...
```

Questa tecnica ha trasformato il kernel da Memory-Bound a Balanced.

I Sub-Steps sono una tecnica di integrazione numerica che aumenta la precisione e la stabilità della simulazione.

Benefici

- La simulazione diventa molto più accurata
- Aumento intensità aritmetica

NVIDIA Nsight Compute	1 Sub-Step	4 Steps	8 Sub-Steps	Miglioramento
% No Eligible Warps	49.24%	34.71%	14,83%	-34.4 punti %
Warp Pronte / Scheduler	1.33 warp	1.84 warp	4.34 warp	+226%
Latenza generale (cicli/istruzione)	20.22 cicli	16.23 cicli	13.23 cicli	-35%
Attesa Media / Istruzione	11.1 cicli	7.0 cicli	2.2 cicli	-81%
L1/TEX Hit Rate	29.30%	33.14%	21.28%	-8.0 punti %

Ottimizzazione con la __shared__ Memory

File: 3_sharedmem.cu

- __global__ → __shared__

```
// All'interno del kernel
extern __shared__ float s_data[];

// Partiziona la memoria condivisa
float* s_posX_ptr = (float*)s_data;
// ... (e gli altri puntatori s_*) ...

// Ogni thread carica i dati della sua particella
s_posX_ptr[threadIdx.x] = p_posX[global_idx];
s_posY_ptr[threadIdx.x] = p_posY[global_idx];
// ... (carica tutti gli altri campi) ...

// Sincronizza per garantire che il caricamento sia completo
__syncthreads();
```

- __shared__ per i registri

```
// I calcoli usano i dati appena caricati dalla shared memory
float l_posX = s_posX_ptr[threadIdx.x];
float l_posY = s_posY_ptr[threadIdx.x];
// ... (carica le altre variabili) ...

#pragma unroll
for (int i = 0; i < SUB_STEPS; ++i) {
    // Tutta la fisica e il respawn operano su queste
    // variabili locali nei registri
}
```

- Registri su __shared__

```
__syncthreads();

s_posX_ptr[threadIdx.x] = l_posX;
//                  ...
```

- __shared__ → __global__

```
__syncthreads();

s_posX_ptr[threadIdx.x] = l_posX;
//                  ...
__syncthreads();

// Dalla shared alla memoria globale
p_posX[global_idx] = s_posX_ptr[threadIdx.x];
```

NVIDIA Nsight Compute	Senza Shared mem	Con Shared mem	Impatto
L1/TEX Hit Rate	21.28%	10.05%	-53%

- L'uso della __shared__ memory come cache manuale
  → Diminuzione dell'utilizzo della cache L1

- Centralizzando tutti gli accessi ai dati in __shared__, il kernel ha potuto eseguire il suo carico di lavoro quasi senza stalli

Feature aggiuntiva: Fuoco e fumo

File: 4_interaction_particles.cu

1) A intervalli di tempo, un evento globale designa un blocco di thread come "epicentro" di una raffica.

```
// All'inizio del kernel
if (threadIdx.x == 0) {
    // Calcola se è ora di un nuovo evento
    if (fmod(time, turbulence_period) < dt) {
        // Scegli un blocco target casuale
        unsigned int targetBlock = ((unsigned int)(time * 100.0f)) % gridDim.x;

        if (blockIdx.x == targetBlock) {
            // Se sono il blocco scelto, calcolo la forza del vento
            s_turb_force_X[0] = ...;
            s_turb_force_Y[0] = ...;
        }
    }
}
__syncthreads();
```

2) Il primo thread di quel blocco calcola una forza di vento e la scrive in __shared__ memory.

```
bool block_has_turbulence = (s_turb_force_X[0] != 0.0f || s_turb_force_Y[0] != 0.0f);

if (block_has_turbulence) {
    my_turbulence_flag = 1.0f;
}
```

3) Le particelle colpite cambiano il loro stato in "fumo", seguendo una fisica e un rendering che le fa apparire più grandi, grigie e trasparenti.

```
if (l_lifetime > 0.0f) {
    // ...
    if (my_turbulence_flag > 0.0f) {
        // --- FISICA DEL FUMO ---
        // Le particelle vengono spinte dalla forza del vento
    }
    else {
        // --- FISICA DEL FUOCO ---
        // Le particelle seguono la logica normale
    }
}
```

Interazione tra thread

NVIDIA Nsight Compute	Fuoco (ottim. + shared)	Fuoco e fumo (ottim. + shared)	Impatto
Tempo di Esecuzione (µs)	197.44 µs	221.50 µs	+12%
No Eligible Warp	13.41%	11.57%	Quasi Identico
Warp Pronte / Scheduler	4.30 warp	4.19 warp	Quasi identico
Latenza Generale (Cicli/Istruzione)	13.23 cicli	13.41 cicli	Quasi identico

Questo approccio ha permesso di creare un nuovo effetto visivo, mantenendo le performance del kernel, che è rimasto bilanciato.

Panoramica finale: prestazioni

NVIDIA Nsight™ Compute	Parallelismo Naive (AoS)	Ottimizzazione Memoria (SoA)	Tuning Occupancy	Aumento Intensità Aritmetica e __shared__	Finale con Interazione (__shared__)	Impatto finale
Latenza Generale (Cicli/Istruzione)	137 cicli	41.2 cicli	20.2 cicli	13.2 cicli	13.4 cicli	-90.2%
Stalli di Memoria (Cicli)	77 cicli	30.8 cicli	11.1 cicli	2.2 cicli	Secondario	Quasi Eliminati
Warp Pronte/Scheduler	0,16 warp	0.28 warp	1.33 warp	4.34 warp	4.19 warp	+2518%
% Tempo Inattivo (No Eligible)	96,12%	83.5%	49.2%	14.8%	15.4%	-80.7 punti %
Tempo esecuzione kernel (media)	697,47 µs	circa 550 µs	258,08 µs	204,61 µs	221,50 µs	-68.2%
Stato del Kernel	Limitato da Latenza e Divergenza	Memory-Bound	Memory-Bound	Balanced (50.8% ALU)	Balanced (48.2% ALU)	Trasformazione Completa

Studio delle Prestazioni su Diversi Dataset

Numero particelle	Tempo Esecuzione Kernel	Warp Pronte/Scheduler	% Tempo Inattivo	Analisi del Comportamento
64K	22,85 µs	2,70 warp	38,33%	Sotto-utilizzata
512K	116,93 µs	3,8 warp	17,93%	Miglioramento
1M	197 µs	4.34 warp	14.8%	Ottimo
2M	435,90 µs	4,26 warp	14,9%	Saturazione
16M	3,39 ms	4,36 warp	12,75%	Saturazione piena

il kernel raggiunge il suo picco di efficienza intorno a 1 milione di particelle, per poi scalare in modo lineare con il tempo d'esecuzione.

Soluzione robusta

Analisi delle prestazioni rendering

Numero particelle	Tempo Esecuzione Kernel	FPS medi	Impatto del Kernel sul Frame Time (%)	Analisi del Comportamento
64K	22,85 µs	144+ FPS	~0.33%	Poco realistico
512K	116,93 µs	133 FPS	~1.55%	Molto fluido, ma il fuoco appare scarno
1M	197 µs	67 FPS	~1.32%	Buon compromesso tra densità e fluidità
2M	435,90 µs	36 FPS	~1.57%	Scattoso, ma l'effetto è denso
16M	3,39 ms	10 FPS	3.39%	Lento

Il tempo di calcolo del kernel CUDA è solo una piccola frazione del tempo totale necessario per renderizzare un fotogramma. → Il collo di bottiglia ora risiede nel rendering di un numero elevato di particelle trasparenti e sovrapposte