



SAPIENZA
UNIVERSITÀ DI ROMA

DIPARTIMENTO DI INFORMATICA

RayTracing in CUDA

PROGRAMMAZIONE EMBEDDED E MULTICORE

Professore:
Pontarelli Salvatore

Studenti:
Tarantelli Kristjan, 2009153
De Nicola Tommaso, 2006686

Indice

1	Introduzione	2
1.1	Ray Tracing	2
1.2	CUDA (Compute Unified Device Architecture)	3
2	Sviluppo	4
2.1	Come funziona un Ray Tracer	4
2.2	Parallelizzazione del Codice	6
3	Analisi e Ottimizzazione	7
3.1	Analisi delle prestazioni	7
3.2	Dimensioni dei blocchi	7
3.3	Shared Memory	8
3.4	Ottimizzazioni della Memoria	9
3.5	Ottimizzazioni Generali	10
3.5.1	Funzioni ricorsive	10
3.5.2	Recupero delle Virtual Tables delle classi	10
3.5.3	Precisione dei Calcoli	10
3.6	Limiti: Warp Divergence	11
3.7	Possibili Soluzioni: Dynamic Parallelism e Blocchi 3D	11
4	Conclusioni	12
	Riferimenti bibliografici	13

1 Introduzione

L'idea del progetto nasce dalla volontà di approfondire il modo di creazione e di manipolazione delle immagini, introdotto durante il corso. L'idea iniziale era di creare un Ray Tracer real-time da testare su un videogioco semplice, come Doom, ma ci siamo subito accorti che sarebbe stata un'impresa troppo grande. Nulla vieta che il progetto potrebbe essere esteso in futuro con più calma.

Ciò che abbiamo fatto, invece, è stato creare un Ray Tracer di immagini statiche ad alta definizione utilizzando le tecniche di parallelizzazione di CUDA. Abbiamo analizzato le prestazioni e cercato di ottimizzarle, ottenendo discreti risultati su alcuni aspetti.

1.1 Ray Tracing

Non conoscendo le basi dietro il funzionamento dei Ray Tracer, abbiamo studiato con molto interesse il libro *Ray Tracing in One Weekend* [1].

La parola Ray Tracing cela, infatti, un vasto insieme di tecniche che sfruttano la geometria ottica per comprendere ed emulare il percorso dei raggi della luce e le loro interazioni con le superfici. Queste tecniche vengono utilizzate sia nella modellazione di sistemi ottici, come le lenti delle fotocamere, sia nella Computer Grafica, per simulare la luce e la riflessione in modo più realistico.

A differenza di altri metodi di rendering, il ray tracing simula il percorso della luce nel modo più fedele possibile. Il processo inizia con la generazione di raggi di luce virtuali che partono dalla sorgente luminosa e vengono tracciati attraverso la scena virtuale. Ogni raggio può interagire con gli oggetti incontrati nel suo percorso, dando luogo a fenomeni come riflessione, rifrazione, ombre e illuminazione globale.

Le fasi principali del Ray Tracing includono:

1. Lancio dei Raggi (Ray Casting): vengono generati raggi dalla sorgente luminosa attraverso ogni pixel dell'immagine;
2. Intersezione dei Raggi: i raggi vengono tracciati attraverso la scena, e quando incontrano un oggetto, viene calcolato il punto di intersezione;
3. Calcolo della Luce: in base alle caratteristiche dei materiali degli oggetti e alle proprietà della luce, vengono calcolati i valori di colore per ogni pixel.
4. Rendering Finale: l'immagine finale viene composta con tutti i calcoli effettuati per ogni pixel, creando un'immagine realistica e dettagliata.

Il ray tracing è noto per produrre risultati visivi eccezionali, ma richiede notevoli risorse computazionali e può essere un processo intensivo in termini di tempo. Tuttavia, con l'avanzare della tecnologia hardware, l'utilizzo del ray tracing sta diventando sempre più diffuso nei settori dell'intrattenimento digitale, offrendo esperienze visive sempre più immersive e realistiche.

1.2 CUDA (Compute Unified Device Architecture)

CUDA è una piattaforma di calcolo parallelo per l'utilizzo delle GPU NVIDIA, progettate apposta per gestire una grande quantità di operazioni simultanee. Poiché il Ray Tracing coinvolge una grande quantità di calcoli parallelizzabili, la loro architettura è particolarmente adatta ad affrontare questa mole di lavoro in modo efficiente.

Come già citato, è noto lo svantaggio in termini di performance degli algoritmi di Ray Tracing rispetto ad altri come lo Scanline Rendering [2] che utilizza la coerenza dei dati per gestire la computazione tra i pixel. La scelta di trattare ogni raggio in modo separato fa ricominciare tutto il procedimento ad ogni nuovo pixel e se da un lato questo offre il vantaggio di migliorare la qualità dell'immagine, dall'altro genera un notevole impiego di risorse.

Il lancio dei raggi, però, può essere eseguito in modo parallelo tramite i CUDA threads assegnando la computazione di un pixel ad ogni thread. Questo approccio consente di gestire scenari più complessi e di aumentare la risoluzione delle immagini senza subire un degrado significativo delle prestazioni.

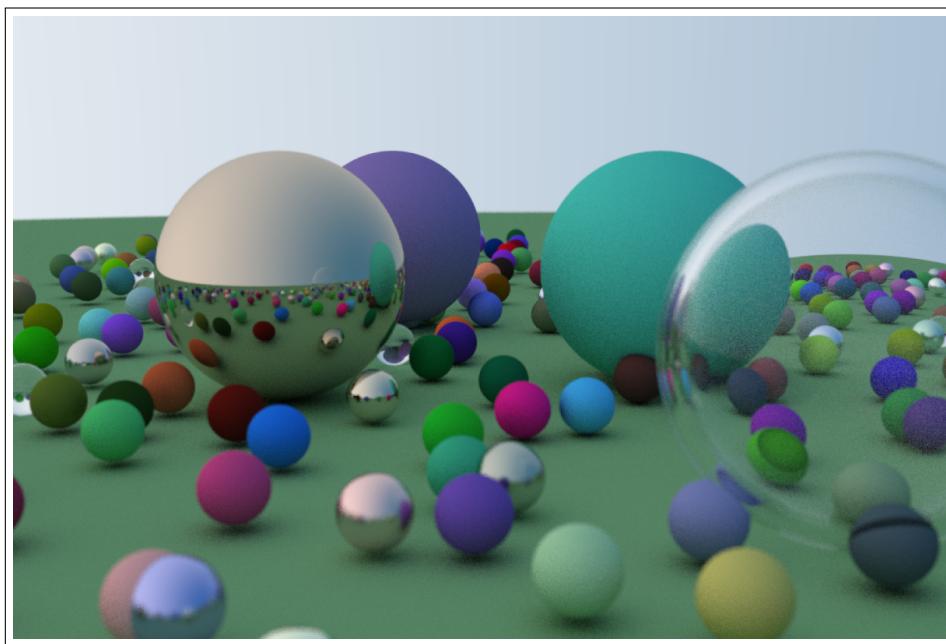


Figura 1: Immagine generata con Ray Tracing

2 Sviluppo

2.1 Come funziona un Ray Tracer

Il sistema di coordinate adottato per il Ray Tracing è comunemente uno spazio tridimensionale, come illustrato nella Figura 2. In questo spazio, il mondo della scena è posizionato nella parte negativa dell'asse z e viene riflessa nella *viewport* attraverso la camera in modo specolare.

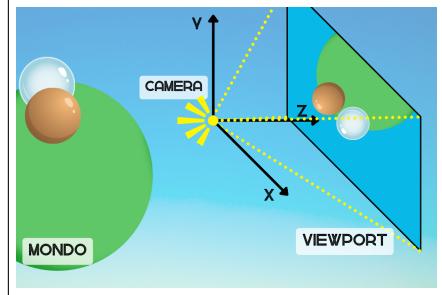


Figura 2: Coordinate Teoriche

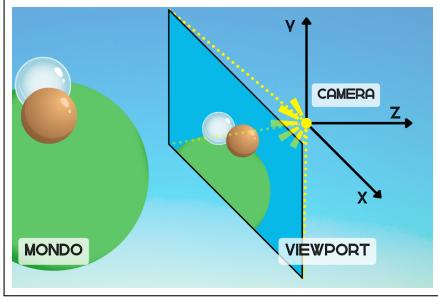


Figura 3: Coordinate Pratiche

Tuttavia, per semplificare i calcoli, nella pratica si sposta la *viewport* nella parte negativa dell'asse z, come mostrato nella Figura 3. In questo contesto, un raggio è rappresentato come un vettore con origine nella camera e direzione lungo il segmento che congiunge il pixel associato al raggio con la camera stessa: $\vec{Ray} = \vec{pxl} - \vec{Cam}$.

Nel codice, la generazione dei raggi è modellata in modo analogo:

```
1 ray r(cam_0, lower_left_corner + u*width + v*height - cam_0);
```

Una volta generato il raggio, che possiamo scrivere come $\vec{R}(t) = \vec{O} + t\vec{D}$ in cui \vec{O} è l'origine e \vec{D} la direzione, si vanno a calcolare le intersezioni con tutti gli oggetti presenti nel mondo, entro un certo intervallo limite. Tra tutti i valori t^* trovati, si considera il minimo, t_{min} , che rappresenta il punto di intersezione più vicino alla camera: $\vec{R}(t_{min})$ è dunque il punto di interesse nello spazio. In base all'oggetto colpito, il raggio si comporta in modo diverso.

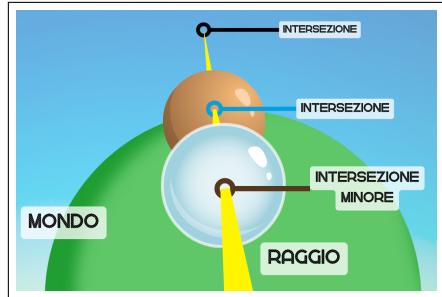


Figura 4: Intersezioni

Gli oggetti possono assumere forme, materiali e colori diversi. La forma determina l'equazione da risolvere nel calcolo dell'intersezione: la più semplice è quella della sfera.

$$(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$$

Sostituendo ad x, y e z le coordinate del raggio variabili in t troviamo l'equazione quadratica da risolvere per trovare l'intersezione

$$(\vec{R}(t) - \vec{C}) \cdot (\vec{R}(t) - \vec{C}) = r^2$$

Il materiale determina, invece, la modalità di diffusione (*scatter*) del raggio che lo ha colpito. Quelli trattati sono:

- Metallo: produce una riflessione speculare del raggio rispetto alla normale dell'oggetto.
- Opaco: produce una riflessione diffusa del raggio, ovvero non lungo la direzione speculare, ma lungo direzioni casuali. Un diffusore ideale è detto lambertiano e riflette la luce omogeneamente in tutte le direzioni.

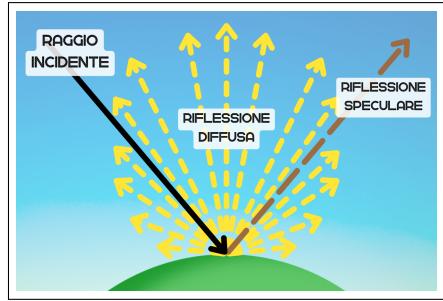


Figura 5: Riflessione

- Vetro: nei dielettrici il raggio si sdoppia in una componente riflessa e una rifratta. Abbiamo semplificato il concetto scegliendo casualmente una delle due componenti da diffondere per evitare di avere tempi di computazione esponenziali.

Infine, il colore determina il potere riflettente di un oggetto (*albedo*). Di fatti, più un oggetto tende al bianco (1,1,1), più tenderà a riflettere i colori degli oggetti vicini; più tende al nero (0,0,0), e più assorbirà i colori. Questo comportamento è ottenuto moltiplicando, per ogni pixel, i colori degli oggetti intersecati dal raggio associato durante le riflessioni e rifrazioni. Da notare che quando il raggio non interseca nulla assorbe il colore dello sfondo, mentre quando rimbalza molte volte tra vari oggetti si genera un colore scuro: un'ombra.



Figura 6: Aliasing

Ora, nonostante sia posta un'alta definizione, si nota una certa discontinuità nell'immagine. Ciò è dovuto all'effetto aliasing, che rende l'immagine ben lontana dalla realtà a cui siamo abituati in cui i contorni sono influenzati dai colori vicini. La tecnica di antialiasing consiste nel campionare (*sampling*) ogni pixel generando più raggi con direzioni lievemente modificate.

Il colore associato a ciascun pixel è la media ponderata dei colori restituiti da questi raggi, rendendo l'immagine più realistica all'aumentare del numero di campioni.

Infine, bisogna considerare la possibilità di spostare la camera ed utilizzare degli effetti avanzati. Lo spostamento della camera viene ottenuto eseguendo dei cambi di coordinate e riposizionando la viewport. Altezza e larghezza di quest'ultima sono dipendenti dal parametro *vfov* (*Vertical Field of View*) che permette di regolare lo zoom. Con i parametri di *apertura* e *distanza focale* è possibile riprodurre la messa a fuoco: maggiore è l'apertura, maggiore sarà la sfocature dell'immagine. Per ottenere questo effetto, durante la generazione di un raggio, anziché prendere come origine il centro della camera, si seleziona un punto casuale in un suo intorno, definito in modo proporzionale all'apertura.

```

1 ray get_ray(...) {
2     vec3 r = rnd_in_disk() * cam->aperture / 2.0f;
3     return (ray(cam->origin + dot(cam->axis, r), ...)); }
```

2.2 Parallelizzazione del Codice

La gestione di una lunga lista di oggetti all'interno del mondo può comportare tempi di esecuzione eccessivamente prolungati. In questo contesto, la parallelizzazione emerge come un elemento chiave, sfruttando la collaborazione tra threads al fine di accelerare l'esecuzione del programma.

Per affrontare questa sfida, abbiamo adottato la *Metodologia Foster* [3], che ci ha permesso di costruire uno schema generale del programma con i seguenti passi:

1. **Partizionamento:** abbiamo identificato le attività suscettibili di parallelizzazione, come la generazione dei raggi, il calcolo delle intersezioni lungo la loro traiettoria e la determinazione del colore associato a un pixel specifico.
2. **Comunicazione:** è necessario che le attività ricevano le informazioni sul mondo e i dati relativi alla camera per eseguire i calcoli. Al termine, è essenziale che comunichino i risultati ottenuti. Inoltre, attività che agiscono sullo stesso raggio devono condividere i dati ad esso relativi.
3. **Agglomerazione:** per ridurre eccessive comunicazioni e ottimizzare le operazioni, è possibile aggregare le attività relative a un singolo raggio. Un'attività può così gestire la generazione del raggio, la diffusione e il calcolo del colore da restituire.
4. **Mappatura:** considerando che le attività da eseguire sono simili e devono essere ripetute su molti dati diversi, viene logico adottare un hardware di tipo *SIMD* (*Single Instruction Multiple Data*) [4] come le GPU. Ognuna di queste attività, infatti, può essere eseguita da un *CUDA thread* tramite un kernel. L'*host* sarà responsabile di inviare i dati corretti e recuperare i risultati ottenuti.

Per implementare queste fasi, ci siamo ispirati a un articolo sul blog *NVIDIA*[5] che tratta le stesse tematiche. Di base, il flusso del programma coinvolge l'inizializzazione delle memorie, la chiamata al kernel di rendering e la copia del buffer dell'immagine prima sull'*host* e successivamente su file.

L'*host* si occupa di allocare sulla memoria del *device* il mondo e la camera. Di conseguenza, il kernel di creazione del mondo viene chiamato per inizializzare la lista di oggetti e la camera direttamente sulla memoria del dispositivo.

Inizia quindi la fase cruciale: viene chiamato il kernel di rendering, assegnando un thread a ogni pixel dell'immagine. Dopo aver calcolato il proprio ID, ogni thread genera un numero prestabilito di raggi, affidandosi ai valori della camera. Il thread calcola l'intersezione con l'oggetto più vicino, tenendo conto della diffusione e del colore colpito. Questi fattori determinano la nuova direzione di propagazione e l'attenuazione del raggio. I colori dei raggi campionati vengono mediati e applicati al buffer dell'immagine. Al termine delle operazioni, la memoria utilizzata viene liberata attraverso un kernel apposito e le classiche funzioni di deallocazione. Infine, l'*host* si occupa di generare l'immagine, copiando su file il buffer restituito dal kernel di rendering.

3 Analisi e Ottimizzazione

3.1 Analisi delle prestazioni

Come ci insegna Donald Knuth [6], *"le ottimizzazioni premature sono la radice di tutti i mali"*. Quindi, per evitare di procedere con ottimizzazioni guidate dal sentimento, è essenziale condurre un'analisi dettagliata dell'esecuzione del codice. Dopo aver scoperto che *nvprof* è stato deprecato e che NVIDIA ha deciso di non supportare la profilazione su *WSL* (*Windows Subsystem for Linux*, ambiente in cui lavoriamo entrambi), siamo in qualche modo riusciti a reperire un report.

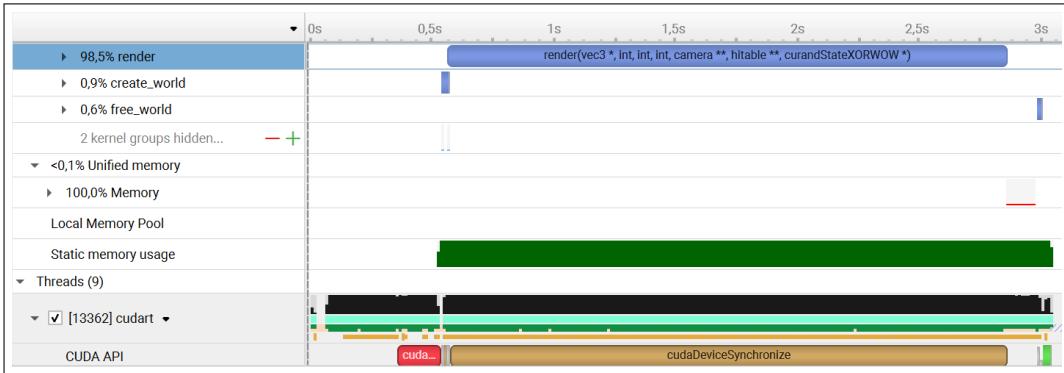


Figura 7: Profilazione tramite Nsight System

N.B.: la profilazione è stata eseguita dopo l'ottimizzazione alle vtables, sezione 3.5.2

Nella sezione iniziale, dedicata all'inizializzazione del profiler, emergono le fasi di allocazione della memoria, prima sull'*host* (indicata in rosso tra le chiamate alle *API CUDA*) e successivamente sul *device* (evidenziata in blu tra le chiamate ai kernel). Conforme alle aspettative, la stragrande maggioranza del tempo di esecuzione è attribuibile al kernel di rendering, rappresentando il 98,5% dei tempi nei kernel e l'87,5% del totale. Pertanto, è cruciale sviluppare strategie mirate a ridurre principalmente il tempo di esecuzione di questo componente.

3.2 Dimensioni dei blocchi

Come prima cosa dobbiamo effettuare una scelta riguardo la dimensione dei blocchi da utilizzare per il kernel. Infatti, in base alle *compute capabilities* della GPU, ci saranno delle configurazioni che sfruttano meglio l'hardware del dispositivo. Questo dipende, inoltre, dall'utilizzo dei registri da parte dei thread del blocco e dalle operazioni che effettua in proporzione alle richieste in memoria: può accadere che il *multi processor* non sia sfruttato a pieno, o che, al contrario, non abbia abbastanza risorse. Inaspettatamente, la dimensione che genera il minor tempo di esecuzione è 32x32.

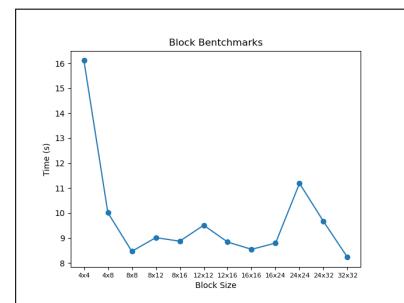


Figura 8: Dimensioni dei Blocchi

3.3 Shared Memory

Notiamo che il tempo di esecuzione dipende principalmente dal numero di oggetti nel mondo, con i quali i raggi devono calcolare l'intersezione minima, e dal numero di raggi emessi per campionare ogni pixel. Ci siamo concentrati soprattutto su quest'ultimo aspetto, attribuibile all'effetto anti-aliasing.

Come già accennato, lo svantaggio prestazionale degli algoritmi di ray tracing è dovuto alla scelta di trattare ogni raggio in modo separato, riavviando il processo di esecuzione ogni volta. Ma cosa accade se mettiamo in collaborazione pixel adiacenti?

L'idea è semplice: invece di far partire numerosi raggi da ciascun pixel, ne emettiamo pochi e sfruttiamo i risultati già calcolati dai pixel adiacenti. Per implementarla, sfruttiamo la *shared memory*.

Ogni thread, corrispondente a un singolo pixel, si occupa di inizializzare la propria cella nella matrice condivisa. Successivamente, calcola la media dei colori restituiti dai raggi generati e la immagazzina nella sua cella di memoria. Solo quando tutti i thread

```

1  __shared__ vec3 sh_mem[B_H][B_W];
2  ... /* threads compute 'color' */
3  sh_mem[tIdx.y][tIdx.x] = color;
4  __syncthreads();
5  vec3 near_col = vec3(0,0,0);
6  near_col += sh_mem[tIdx.y][tIdx.x+1];
7  near_col += sh_mem[tIdx.y][tIdx.x-1];
8  near_col += sh_mem[tIdx.y+1][tIdx.x];
9  near_col += sh_mem[tIdx.y-1][tIdx.x];
10 color = (1-w)*color + w*near_col / 4;
```

del blocco, che condividono la matrice, hanno completato il processo, inizia lo scambio di dati: ogni thread acquisisce i risultati dei pixel adiacenti dalla matrice e ne calcola la media. Infine, effettua una media ponderata tra il suo risultato e il calcolo appena effettuato, determinando il colore da assegnare al pixel.

Si nota che un'immagine con circa 60 campioni è quasi indistinguibile da un'immagine con 25 campioni generata mediante l'approccio descritto. Inoltre, come previsto, la riduzione dei campioni da generare comporta una significativa diminuzione dei tempi di esecuzione.

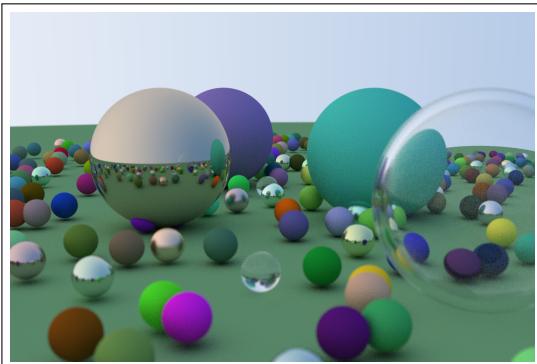


Figura 9: no Sh.Mem., 60 campioni.

Tempo di esecuzione: 5.4s

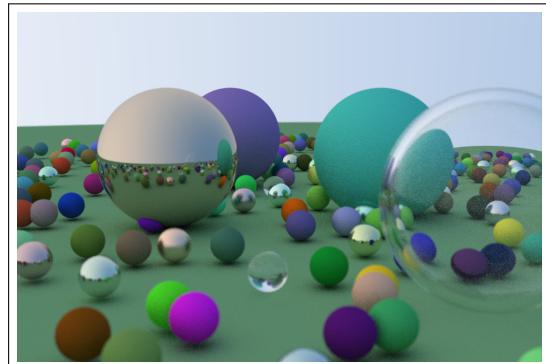


Figura 10: con Sh.Mem., 30 campioni.

Tempo di esecuzione: 2.7s

3.4 Ottimizzazioni della Memoria

Se torniamo all’analisi dell’esecuzione del programma, visibile in Figura 7, possiamo porre l’attenzione sul colore rosso, che in questo caso indica la memoria. La chiamata a funzione *cudaMallocManaged* e i *Page Fault* che si riscontrano a fine programma, evidenziati nella Figura 11, costituiscono il secondo evento più rilevante nell’analisi, occupando più del 10% della totale esecuzione del programma.

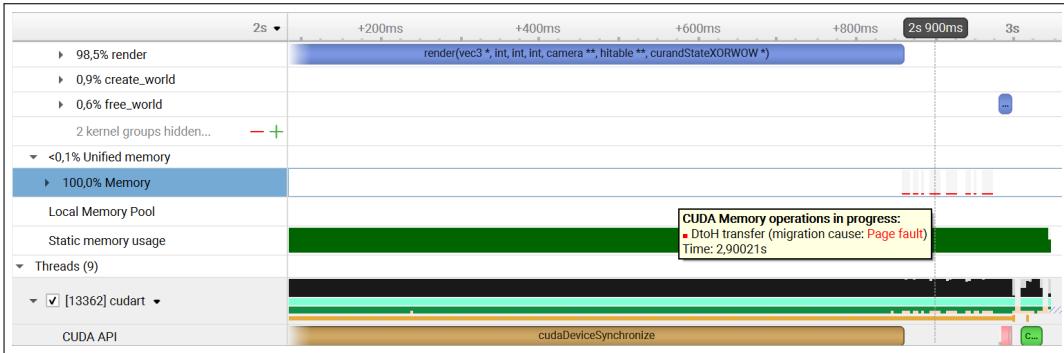


Figura 11: Analisi della Memoria con Nsight System

Questo è dovuto all’utilizzo della cosiddetta *Unified Memory*, che permette di mantenere la coerenza tra le memorie di *host* e *device* senza eseguire copie esplicite. Infatti, appena avviene una sincronizzazione al termine di un kernel, vengono riportate automaticamente le modifiche sulle memorie. Se da un lato facilita la scrittura di codice, dall’altro l’utilizzo della *Unified Memory* può creare problemi prestazionali. Abbiamo deciso quindi di effettuare le copie in modo esplicito, sfruttando la funzione *cudaMallocHost*, che permette di allocare i dati direttamente sulle pagine *pinned*, e la funzione *cudaMemcpy*.

```

1 // cudaMallocManaged((void **) &d_buf, BSIZE);
2 cudaMallocHost((void **) &h_buf, BSIZE));
3 cudaMalloc((void **) &d_buf, BSIZE));
4 cudaMemcpy(d_buf, h_buf, BSIZE, cudaMemcpyHostToDevice));
5 /* kernel di rendering */
6 cudaMemcpy(h_buf, d_buf, BSIZE, cudaMemcpyDeviceToHost);

```

Nonostante la creazione del mondo non costituisca un problema dal lato delle prestazioni, abbiamo voluto provare un approccio diverso per avere il controllo del mondo anche lato *host*. In particolare, abbiamo provato ad allocare gli oggetti prima sull’*host*, per poi copiarli sul *device*. Avendo una lista di oggetti, abbiamo provato a seguire il pattern della copia di *Array of Struct*, con l’idea di migliorarla successivamente con il pattern di *Array of Struct*. Ci siamo accorti, però, che gli oggetti utilizzati sono ben più complessi di semplici strutture dati. Infatti, i dati che contengono sono spesso puntatori ad oggetti di altre classi, andando a formare una catena di dipendenze difficile da sciogliere.

3.5 Ottimizzazioni Generali

Per migliorare ulteriormente le prestazioni, possiamo ridurre alcune operazioni che risultano essere computazionalmente onerose, come l'utilizzo di funzioni ricorsive, il recupero delle *VTables* (*Virtual Tables*) delle classi e l'eccessiva precisione nei calcoli.

3.5.1 Funzioni ricorsive

La funzione responsabile di calcolare il colore per un raggio è stata implementata inizialmente in modo ricorsivo per migliorare la comprensione del codice. La funzione si occupava di calcolare il punto d'intersezione e la nuova direzione del raggio, per poi chiamarsi ricorsivamente utilizzando i parametri appena calcolati. Lo stesso risultato è stato raggiunto attraverso l'utilizzo di un'analogia funzione iterativa.

3.5.2 Recupero delle Virtual Tables delle classi

Il recupero delle *VTables* è demandato alla risoluzione della chiamata a funzione in fase di runtime, anziché al compilatore. Questo processo può essere considerato lento soprattutto in base all'utilizzo delle classi nel codice. Nel nostro contesto, le funzioni critiche sono principalmente quelle dedicate al *sampling* e al calcolo delle collisioni. Pertanto, abbiamo introdotto varianti delle funzioni originali, strutturate in modo tale da non appartenere a classi.

Questa scelta ci ha permesso di ottenere notevoli miglioramenti nei tempi di esecuzione, ma non è stata estesa a tutte le situazioni: ad esempio, nella diffusione dei materiali, riprodurre l'ottimizzazione avrebbe implicato di dover effettuare diversi controlli per selezionare la corretta funzione da chiamare; abbiamo quindi deciso di non farlo. C'è da sottolineare che, mentre questa operazione di ottimizzazione può essere estremamente efficace in determinate situazioni, potrebbe rivelarsi altrettanto inefficace in altre, a seconda delle caratteristiche specifiche di implementazione e utilizzo nel codice.

3.5.3 Precisione dei Calcoli

Le moderne GPU raggiungono le massime prestazioni quando eseguono calcoli in precisione singola, mentre i calcoli a doppia precisione possono risultare notevolmente più lenti. Pertanto, è cruciale fare attenzione durante la costruzione di espressioni con numeri decimali: è sempre buona pratica aggiungere il suffisso '*f*' per evitare che il compilatore interpreti il dato come un numero *double*.

Inoltre, è possibile sacrificare ulteriormente la precisione dei calcoli abilitando in fase di compilazione l'utilizzo delle librerie di *fast math* tramite il *flag* dedicato. Questa rende le operazioni di addizione, moltiplicazione, divisione e radice quadrata molto più veloci, approssimandone i risultati. Non abbiamo notato una visibile perdita nella qualità dell'immagine generata.

3.6 Limiti: Warp Divergence

La divergenza del warp si manifesta quando i thread all'interno di un warp intraprendono percorsi di esecuzione diversi a causa di costrutti di controllo condizionale, come istruzioni *if* o *switch*. Questi thread sono vincolati a seguire il medesimo flusso di codice, per cui, anche se solo uno di essi si immette in una branca condizionale, tutti gli altri sono costretti a seguirla e ad attendere il suo completamento.

La divergenza rappresenta un noto problema nel contesto del Ray Tracing [7], poiché costituisce una caratteristica intrinseca del comportamento dei raggi. La casualità associata alle riflessioni e rifrazioni genera spesso situazioni di divergenza, soprattutto nei pressi dei contorni degli oggetti, in cui alcuni raggi vengono diffusi mentre altri no. La *Warp Divergence* incide negativamente sulle prestazioni complessive del Ray Tracer in quanto comporta una sottoutilizzazione delle risorse della GPU. Tuttavia, essa ha contribuito nello sviluppo di nuovi algoritmi innovativi incitando la ricerca di nuove idee e l'utilizzo di nuove tecnologie per ovviare al problema.

3.7 Possibili Soluzioni: Dynamic Parallelism e Blocchi 3D

Un approccio per mitigare la divergenza dei warp consiste nell'utilizzo della parallelizzazione dinamica, che comporta il lancio di kernel, detti figli, a partire da altri kernel, detti padre. Questo approccio, nel nostro contesto, si può applicare alla fase di rendering: ogni thread lancia un nuovo kernel in modo che ognuno dei thread figli sia responsabile della computazione di un singolo campione.

In primo luogo, si verifica un notevole aumento del numero di thread schedulati, che si traduce sì in maggior parallelizzazione, ma anche in maggior overhead di schedulazione. In secondo luogo, emerge un problema di sincronizzazione, poiché è necessario raccogliere tutti i risultati per poter di procedere. Dato che i kernel sono di base asincroni, per fare questo è richiesta una sincronizzazione manuale, deprecata e quindi da forzare in fase di compilazione tramite la flag `-DCUDA_FORCE_CDP1_IF_SUPPORTED` [8]. Tuttavia, questo approccio comporta uno stress eccessivo sulla GPU, causando un rallentamento significativo rispetto ai metodi precedenti dovuto continue sincronizzazioni.

Un secondo metodo per ridurre la divergenza dei warp consiste nella pianificazione di blocchi tridimensionali, che considerano il numero di campioni come grandezza aggiuntiva. Ciò consente di ridurre notevolmente i costi di sincronizzazione, ma fa sorgere due nuovi problemi: primo, deve essere presente un thread master per ogni pixel che si occupi di riunire i dati restituiti dagli altri e questo genera divergenza; secondo, il numero di campioni è limitato dalle *compute capabilities* della GPU, le quali stabiliscono un massimo di 1024 thread per blocco.

In definitiva, pur ottenendo un lieve aumento delle prestazioni, il costo associato a questo approccio risulta comunque più elevato rispetto alle tecniche descritte nelle sezioni precedenti.

4 Conclusioni

Con questo progetto abbiamo voluto esplorare applicazioni reali della parallelizzazione nel contesto della generazione di immagini. Ci siamo divertiti nel creare mondi a nostro piacimento e testare i limiti della computazione con le GPU.

Partendo da un codice sequenziale che implementa un algoritmo di Ray Tracing, abbiamo applicato i paradigmi della progettazione di costrutti paralleli. Abbiamo quindi analizzato ed ottimizzato il codice sotto vari punti di vista, partendo dall'ottimizzazione del numero di thread utilizzati nei blocchi, fino ad arrivare a questioni più generiche, come la precisione dei calcoli. Abbiamo affrontato i limiti dell'algoritmo parallelizzato e proposto idee per eventuali soluzioni.

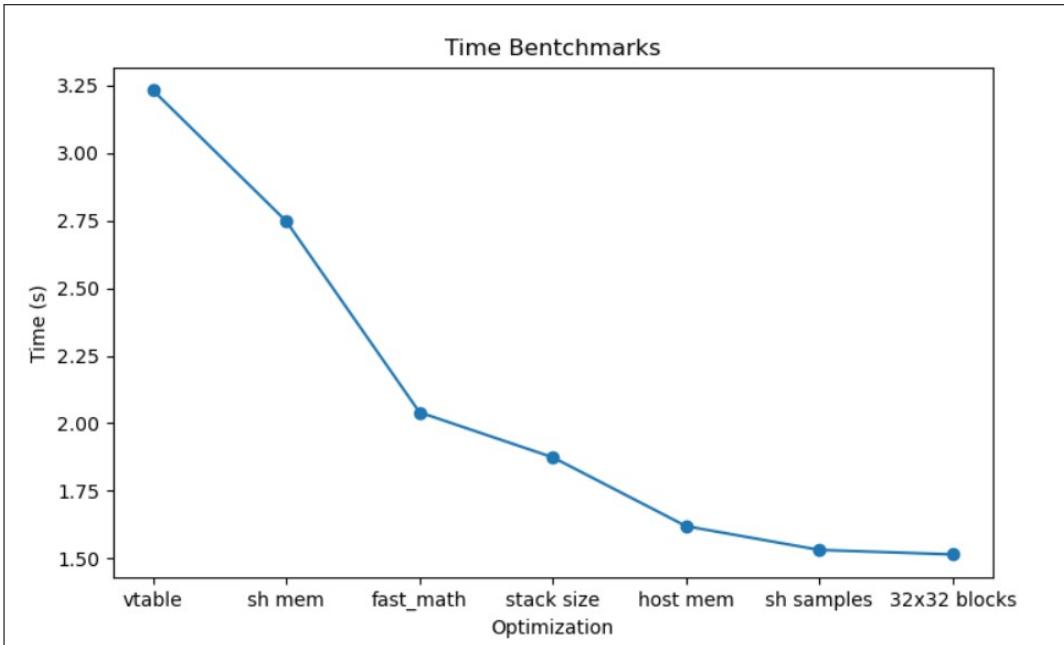


Figura 12: Grafico sui Tempi di Esecuzione con le Ottimizzazioni

Nel complesso, siamo partiti da un codice sequenziale che genera l'immagine in quasi 30min e siamo riusciti ad ottenere un codice parallelo che esegue gli stessi calcoli in poco più di 1min. A seguito di analisi e varie ottimizzazioni siamo riuciti a diminuire in maggior modo il tempo di esecuzione, portandolo a 1.5s. Calcoliamo gli *Speed-up*:

1. Dal codice sequenziale a quello parallelo:

$$\text{Speed-up} = \frac{T_{sequenziale}}{T_{parallelo}} = \frac{29\text{min } 29\text{s}}{1\text{m } 11\text{s}} = 24.9$$

2. Dal codice sequenziale a quello ottimizzato:

$$\text{Speed-up}_{opt} = \frac{T_{sequenziale}}{T_{parallelo_{opt}}} = \frac{29\text{min } 29\text{s}}{1.5\text{s}} = 1179.3$$

Riferimenti bibliografici

- [1] Steve Hollasch Peter Shirley, Trevor David Black.
Ray Tracing in One Weekend. August 2023.
<https://raytracing.github.io/books/RayTracingInOneWeekend.html>.
- [2] Wikipedia. Scanline rendering. https://en.wikipedia.org/wiki/Scanline_rendering.
- [3] Mathematics and Computer Science ANL. Metodologia foster.
<https://www.mcs.anl.gov/itf/dbpp/text/node15.html>.
- [4] Michael J. Flynn. Very High-Speed Computing Systems, volume 54. December 1966.
- [5] NVIDIA Technical Blog. Ray tracing in cuda.
<https://developer.nvidia.com/blog/accelerated-ray-tracing-cuda/>.
- [6] Donald E. Knuth. Structured Programming with go to Statements. ACM Press/Addison-Wesley Publishing Co., New York, 1974.
- [7] Hansung Kim Angie Wang Sizhuo Zhang Yakun Sophia Shao. Cost of Divergence in Ray Tracing: Performance Characterization on CPU and GPU. Sejong University.
- [8] NVIDIA. Nvidia Docs. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/>.