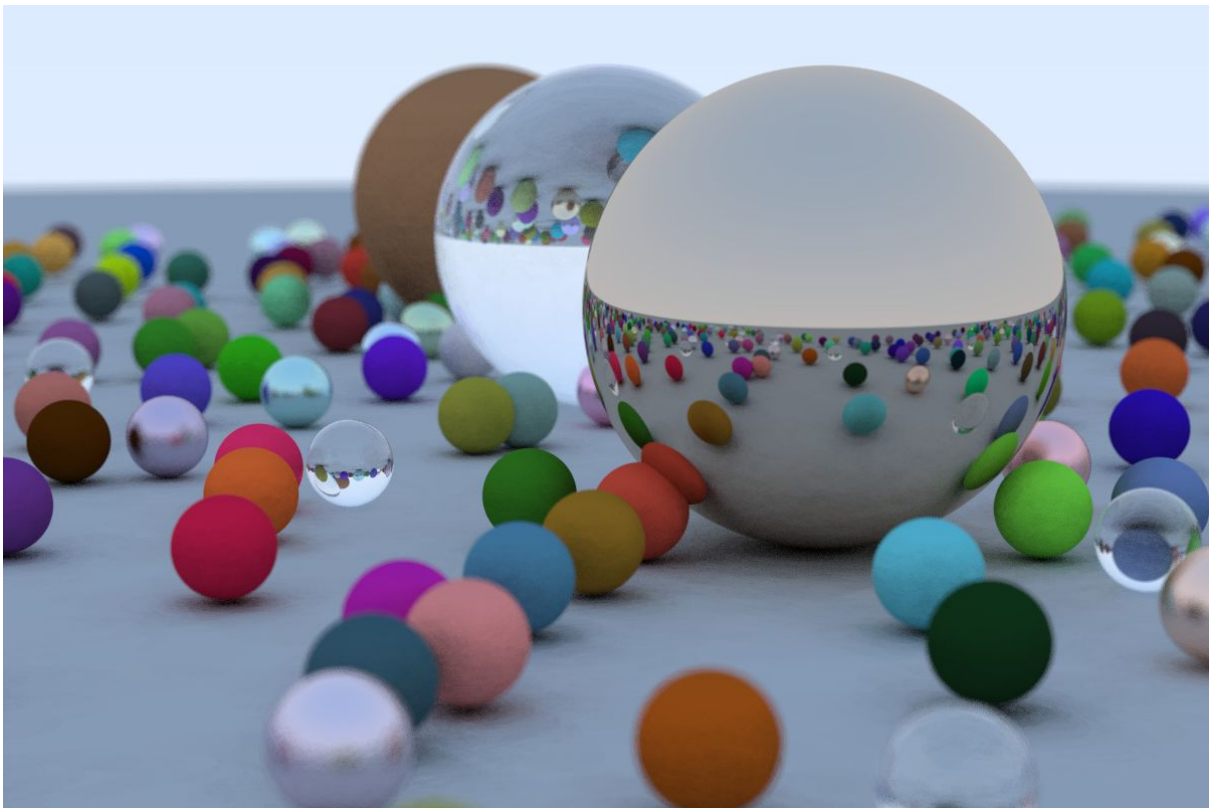


Ray Tracing con CUDA

di Giacomo Garbin



un progetto per il corso di GPU Computing

Università degli Studi di Milano

Introduzione

L'obiettivo di questo progetto è la parallelizzazione di un algoritmo sequenziale. L'algoritmo oggetto dello studio è il ray tracer presentato da Peter Shirley nel suo libro *Ray Tracing in One Weekend*.

In questa relazione verrà innanzitutto presentato brevemente l'algoritmo sequenziale e verranno misurate le sue prestazioni. Dopodiché verrà discusso il processo di parallelizzazione dell'algoritmo e saranno evidenziate le difficoltà emerse nel processo. Infine saranno misurate le prestazioni dell'algoritmo parallelo in diverse configurazioni.

L'algoritmo sequenziale

In computer grafica, il ray tracing è una tecnica di rendering per la generazione di un'immagine bidimensionale di una scena tridimensionale mediante la simulazione dell'interazione tra la luce e gli oggetti nella scena.

L'algoritmo presentato da Peter Shirley nel suo libro *Ray Tracing in One Weekend* è implementato mediante il linguaggio C++ e sfrutta la programmazione ad oggetti offerta dal linguaggio.

Poiché il linguaggio CUDA C che verrà utilizzato nella versione parallela dell'algoritmo non dispone della OOP, è conveniente (e sarà necessario) in questa fase tradurre l'algoritmo in C.

Il codice sorgente dell'applicazione è distribuito nei seguenti moduli.

<i>header.h</i>	contiene la dichiarazione di tutte le strutture e delle funzioni definite nei diversi moduli ad eccezione delle funzioni definite in <i>ray_tracer_cpu.c</i>
<i>vec3.c</i>	contiene la definizione di svariate funzioni per il calcolo vettoriale, come ad esempio la somma o il prodotto scalare tra due vettori
<i>ray.c</i>	contiene la definizione delle funzioni per la gestione del raggio di luce su cui si basa l'algoritmo di ray tracing
<i>sphere.c</i>	contiene la definizione delle funzioni necessarie al calcolo del punto di intersezione tra il raggio di luce e gli oggetti nella scena
<i>material.c</i>	contiene la definizione delle funzioni necessarie al calcolo del colore riflesso di un oggetto colpito dal raggio di luce, il colore dipende dal tipo e dai parametri del materiale dell'oggetto

camera.c contiene la definizione delle funzioni per la configurazione del punto di vista dell'osservatore della scena

ray_tracer_cpu.c definisce l'entry point dell'applicazione, la struttura della scena renderizzata e la sequenza di passaggi eseguiti dall'algoritmo per il calcolo del colore di ciascun pixel dell'immagine generata

La compilazione del codice sorgente avviene con la seguente istruzione

```
nvcc ray_tracer_cpu.c vec3.c ray.c sphere.c material.c camera.c -o ray_tracer_cpu
```

Il risultato prodotto dall'algoritmo può essere registrato con l'istruzione seguente

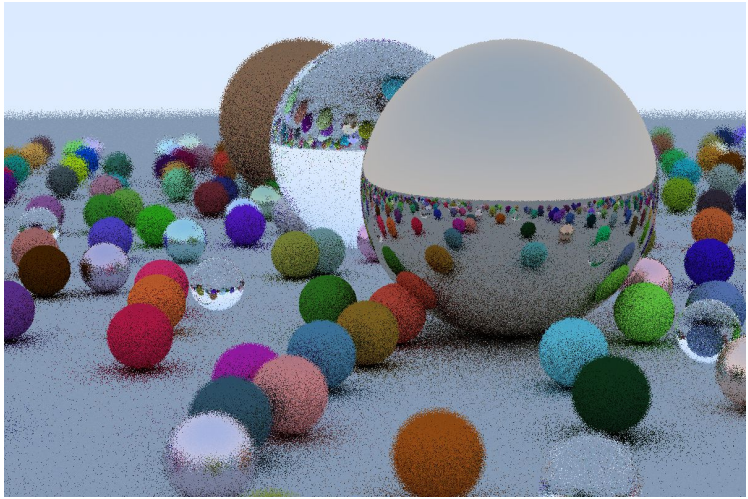
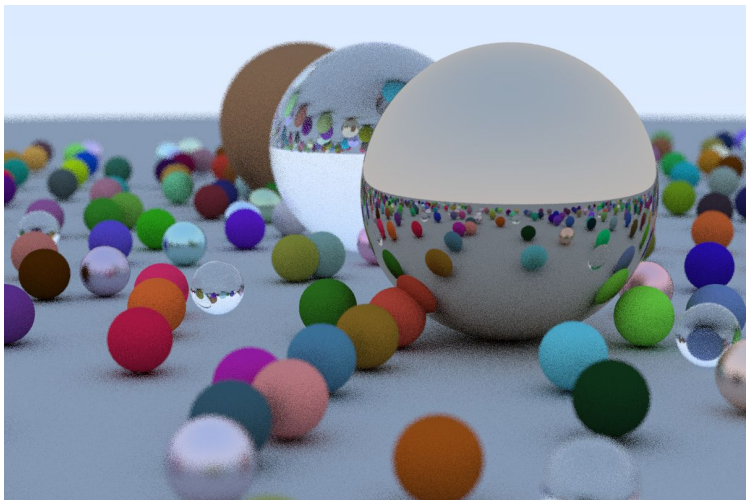

```
./ray_tracer_cpu > image_cpu.ppm
```

L'immagine viene generata in formato PPM ed ha una dimensione di 1,200 pixel in larghezza e 800 pixel in altezza.

```
P3                # "P3" means this is a RGB color image in ASCII
1200 800          # "1200 800" is the width and height of the image in pixels
255              # "255" is the maximum value for each color
219 234 255
219 234 255      # The part below is image data
219 234 255      # (RGB triplets) in row-major order
...
```

Il numero totale di pixel di cui l'applicazione calcola in modo sequenziale il colore è 960,000. Inoltre per ridurre l'effetto di aliasing, il colore di ciascun pixel viene ottenuto come la media aritmetica su un certo numero di campioni. Il numero totale di iterazioni compiute dal ciclo principale del programma è perciò pari a $960,000 * sn$, dove sn è il numero di campioni generati per ciascun pixel.

Le prestazioni dell'algorithm sequenziale vengono misurate con il comando linux *time*. I dati raccolti per diversi valori del numero di campioni utilizzati sono di seguito riportati.

samples	time	image
1	2m 34.067s	
10	25m 35.639s	
100	255m 56.438s	

L'algoritmo parallelo

Le specifiche della scheda grafica utilizzata nella fase di sviluppo e di test dell'algoritmo parallelo sono le seguenti.

<i>model</i>	NVIDIA Tesla M2090
<i>micro-architecture</i>	Fermi
<i>compute capability</i>	2.0
<i>year launch</i>	2011
<i>CUDA cores</i>	512 (16 SMs * 32 CUDA cores)
<i>memory</i>	6 GB on-board global memory 768 KB L2 cache (shared by all 16 SMs) 64 KB on-chip configurable memory (partitioned between shared memory and L1 cache)
<i>max threads per SM</i>	1536 (48 warps per SM)
<i>max grid size</i>	65535, 65535, 65535
<i>max block size</i>	1024, 1024, 64
<i>max threads per block</i>	1024
<i>registers per block</i>	32768
<i>registers per SM</i>	32768
<i>registers per thread</i>	63
<i>max blocks per SM</i>	8

Il codice sorgente dell'algoritmo parallelo è distribuito nei seguenti moduli.

header.h
vec3.cu
ray.cu
sphere.cu
material.cu
camera.cu
random.cu
ray_tracer_gpu.cu

Ciascun modulo rappresenta il porting in versione CUDA C del rispettivo modulo C dell'algoritmo sequenziale, con l'unica aggiunta del modulo *random.cu* per la generazione di punti casuali all'interno dello spazio della scena.

La compilazione del codice sorgente avviene con le seguenti istruzioni.

```
nvcc --device-c ray_tracer_gpu.cu vec3.cu ray.cu sphere.cu material.cu camera.cu random.cu
nvcc ray_tracer_gpu.o vec3.o ray.o sphere.o material.o camera.o random.o -o ray_tracer_gpu
```

La funzione `color_recursive` all'interno del modulo `ray_tracer_gpu.cu` calcola il colore di ciascun pixel in modo ricorsivo, richiamando se stessa al più 50 volte. Questo approccio non crea problemi nella versione sequenziale dell'algoritmo, ma nella versione parallela genera dapprima un warning da parte del linker.

```
nvlink warning: stack size for entry function cannot be statically determined
```

Seguito da un errore se si tenta di lanciare ugualmente il kernel.

#77 cudaErrorIllegalAddress - an illegal memory access was encountered

The device encountered a load or store instruction on an invalid memory address. This leaves the process in an inconsistent state and any further CUDA work will return the same error. To continue using CUDA, the process must be terminated and relaunched.

Il problema viene risolto se si sostituisce la funzione ricorsiva con la funzione equivalente iterativa `color_iterative`.

La configurazione di esecuzione del kernel - block e grid size - può essere specificata mediante i parametri dell'applicazione nel modo seguente.

```
./ray_tracer_gpu X Y > image_gpu.ppm

dim3 block(X, Y);
dim3 grid((nx+block.x-1)/block.x, (ny+block.y-1)/block.y);
...
kernel<<<grid, block>>>();
```

Con un block size 32, 32 si ottiene il seguente errore.

#7 cudaErrorLaunchOutOfResources - too many resources requested for launch

This indicates that a launch did not occur because it did not have appropriate resources. Although this error is similar to cudaErrorInvalidConfiguration, this error usually indicates that the user has attempted to pass too many arguments to the device kernel, or the kernel launch specifies too many threads for the kernel's register count.

Di seguito vengono riportati i tempi di esecuzioni dell'applicazione con diverse configurazioni. Si noti che il numero di campioni per l'anti-aliasing per ciascun pixel è fisso a 100.

block size	time / error
32, 32	cudaErrorLaunchOutOfResources
32, 16	1m 59.171s
16, 16	1m 47.124s
16, 8	1m 37.307s
8, 8	1m 34.112s
32, 8	1m 47.442s
64, 8	2m 9.892s
128, 1	1m 53.889s
256, 1	2m 33.139s
64, 1	1m 47.982s
32, 1	2m 31.262s
16, 1	3m 54.243s
8, 1	6m 19.293s
4, 4	3m 46.869s
8, 4	2m 18.196s

Conclusioni

Lo *speed up* ottenuto tra la versione sequenziale dell'applicazione e quella parallela è notevole. A parità di qualità dell'immagine - 100 campioni per ciascun pixel - si passa da 255m 56.438s (oltre 4.25 ore) dell'algoritmo sequenziale, ad un miglior tempo osservato di 1m 34.112s dell'implementazione parallela.

Bibliografia

Professional CUDA C Programming, John Cheng & Max Grossman & Ty McKercher

Ray Tracing in One Weekend, Peter Shirley

The C Programming Language, Brian Kernighan & Dennis Ritchie