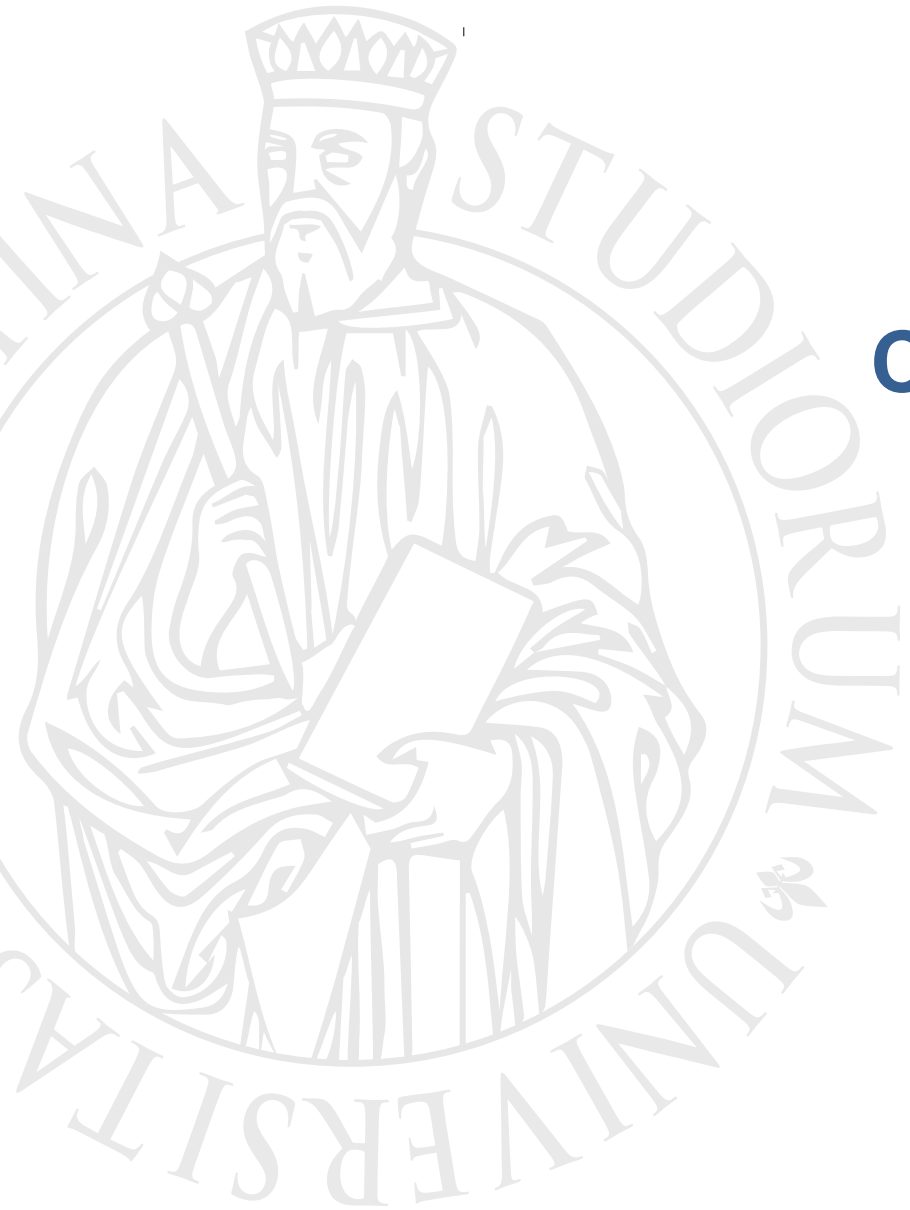


CUDA Circle Renderer

Rendering di figure circolari mediante parallelizzazione NVidia CUDA

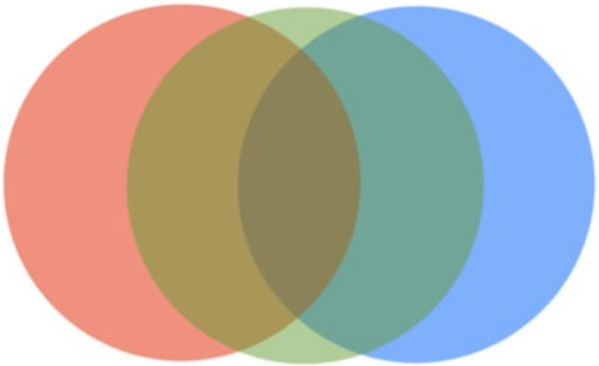
Andrea Spitaleri

A.A. 2019/2020

Disegnare cerchi colorati su schermo

Analisi del processo di rendering

Lo scopo di questo elaborato è quello di mostrare le differenze tra rendering con CPU e rendering con GPU NVidia tramite librerie CUDA, analizzando il caso specifico di rendering di cerchi colorati semitrasparenti.
Prenderemo in considerazione le differenze nelle dinamiche del processo di rendering, e nei tempi richiesti da CPU e GPU per completare lo stesso task che porta ad ottenere l'immagine finale.



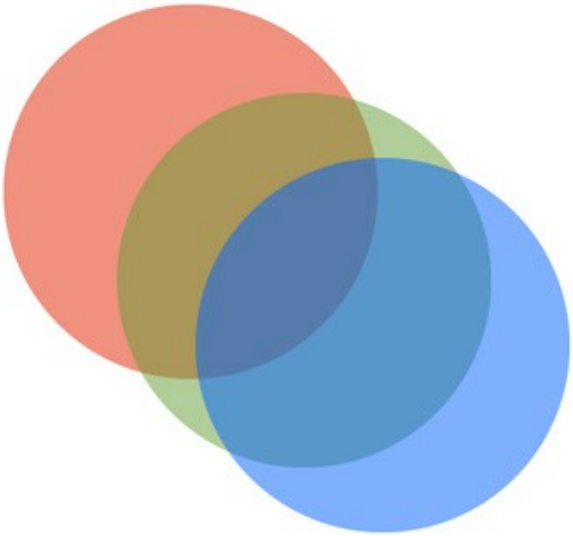
Il rendering in dettaglio

Descrizione degli obiettivi

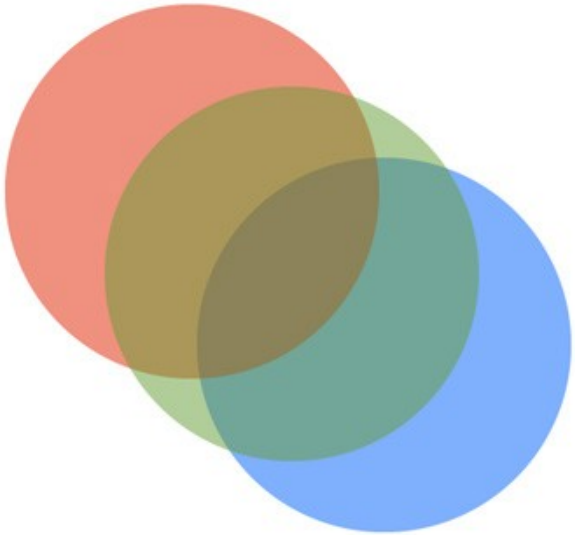
In questo elaborato vogliamo disegnare cerchi colorati sullo schermo. I cerchi hanno diversi colori (quindi diverse componenti di Red, Green, e Blue), dimensione, e posizione tridimensionale nell'immagine (pos. orizzontale, pos. verticale, e profondità).
I cerchi hanno inoltre la caratteristica di essere semitrasparenti. Quest'ultimo fatto introduce un livello di difficoltà di non triviale soluzione nel rendering dell'immagine finale.
Si ha infatti che il colore di un pixel è il risultato di un blending dei colori dei cerchi che lo comprendono, e questo blending è di fatto una funzione non commutativa, in quanto il colore risultante da un cerchio A davanti a un cerchio B è diverso da quello risultante dal cerchio B davanti al cerchio A (come possiamo notare nella successiva diapositiva).

Il rendering in dettaglio

Descrizione degli obiettivi



Correct order:
blue over green over red



Incorrect order

Il rendering in dettaglio

Descrizione degli obiettivi

Questa difficoltà risiede nel fatto che il colore finale di un pixel è dato al 50% dal colore del cerchio in primo piano, e al 50% dal risultato del blending dei cerchi rappresentati dietro di esso, in modo ricorsivo.
C'è quindi una vera e propria data dependency tra cerchi con coordinate x,y simili.

Il problema non sussiste in caso di rendering Single-Thread in quanto semplicemente andando a "disegnare" i cerchi in ordine dal più lontano, via via fino a quello in primo piano, si ottiene l'immagine corretta.
Nel caso di rendering Multi-Thread le cose si complicano non poco, in quanto questa dipendenza tra dati rende difficile una parallelizzazione della computazione.

Il problema nel caso Multi-Thread

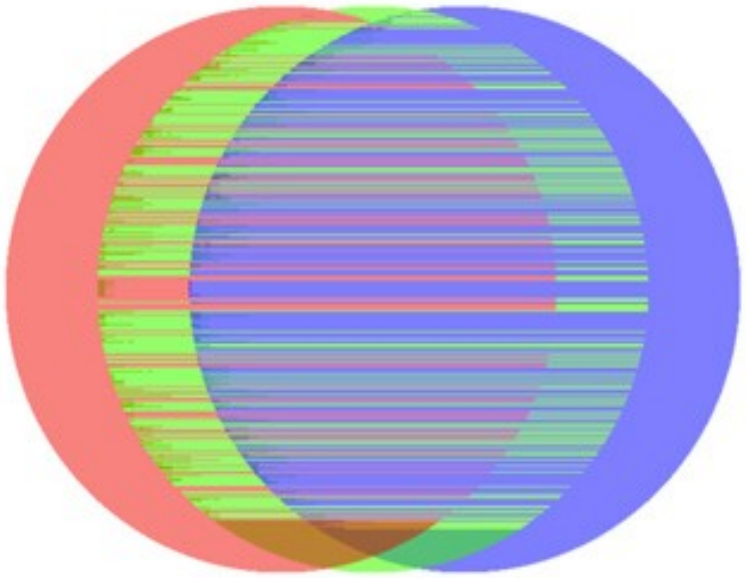
Parallelizzazione con data dependency

Al fine di ottenere un corretto rendering dell'immagine finale occorre garantire due invarianti:
- Atomicità delle operazioni di modifica colore nella matrice di pixel, al fine di evitare race conditions
- Ordinamento di rendering: per cerchi con coordinate simili (e che quindi si sovrappongono) è necessario disegnare prima quelli dietro, poi via via quelli davanti

Nel caso Single-Thread ciò è di banale soluzione, in quanto il primo punto non costituisce un problema per la natura stessa della computazione, e il secondo punto è risolvibile andando a disegnare i cerchi in ordine decrescente di profondità.
Nel caso Multi-Thread invece queste invarianti sono garantibili soltanto attraverso un non-triviale processo di rendering, che analizzeremo in profondità.
Il non rispetto di queste invarianti porta quindi, nel caso Multi-Thread, a ottenere un rendering finale non corretto, ciò a causa di race conditions.

Il problema nel caso Multi-Thread

Parallelizzazione con data dependency



Rendering soggetto a race conditions

I diversi approcci

Single-Thread Sequential Computing vs Multi-Thread Parallel Computing

Nel caso Single-Thread sequenziale il rendering segue la seguente dinamica:

Clear immagine
Per ogni cerchio
    Calcola coordinate del quadrato che comprende tale cerchio
    Per ogni pixel in tale quadrato
        Calcola la coordinata continua del pixel (punto centrale – vedi Point Sampling)
        Se il punto centrale del pixel è dentro il cerchio
            Calcola colore del cerchio nel punto
            Fai il blending di tale colore col colore preesistente nel pixel

I diversi approcci

Single-Thread Sequential Computing vs Multi-Thread Parallel Computing

Nel caso Multi-Thread parallelo CUDA l'immagine finale è divisa in blocchi di pixel associati 1:1 a blocchi di threads. Il rendering, in ogni singolo thread di ogni blocco segue la seguente dinamica:

Clear immagine
Per ogni cerchio in un segmentino dell'array di cerchi (cosicché ogni blocco avrà analizzato l'intero array)
    Se il quadrato che comprende il cerchio ha almeno un punto dentro il blocco corrente
        Salvo il suo indice nella memoria shared condivisa a livello di blocco.
Adesso per ogni blocco c'è una lista degli indici dei cerchi il cui quadrato che lo comprende ha almeno un punto in tale blocco
Per ogni cerchio nella lista di cerchi associati al blocco
    Se il cerchio ha effettivamente almeno un punto nel blocco
        Calcola colore del cerchio nel punto
        Fai il blending di tale colore col colore preesistente nel pixel

Questione di invarianti

In che modo le invarianti sono garantite nel caso Multi-Thread

Nel caso Multi-Thread, al fine di non avere race conditions, dobbiamo garantire le invarianti accennate poco fa:

- Atomicità delle operazioni di modifica colore nella matrice di pixel, al fine di evitare race conditions.
- Ordinamento di rendering: per cerchi con coordinate simili (e che quindi si sovrappongono) è necessario disegnare prima quelli dietro, poi via via quelli davanti.

Nell'implementazione che esamineremo:
- il primo punto è garantito dal fatto che non si ha contention sui dati di un pixel, in quanto ogni thread si occupa solo e soltanto di un singolo pixel.
- il secondo punto è garantito dal fatto che i cerchi vengono passati al renderer già in ordine di profondità e il riorganizzamento in liste realtive ai singoli blocchi viene fatto conservando questo ordine e quindi l'invariante.

Il programma realizzato

Funzionalità del software

L'elaborato è disponibile alla pagina https://github.com/spita90/CUDA-Circle-Renderer, ed è stato realizzato con Qt Creator in linguaggi C++ e CUDA-C.
Senza specificare parametri nella command-line, il programma si apre mostrando tutte le possibilità. Sono le seguenti:

- -r <cpu/cuda> — Consente di selezionare il tipo di renderer che sarà utilizzato
- -s [size] — Consente di specificare in pixel il lato dell'immagine quadrata finale (default 1024, quindi immagine 1024x1024)
- -b — Avvia la modalità benchmark, dove viene fatto sia con CPU che con GPU, per 5 volte il rendering della stessa immagine finale. Viene infine fatto un confronto delle medie dei tempi di rendering.
- -m <good/bad> — Consente di scegliere di utilizzare l'implementazione dell'algoritmo di rendering parallelo da utilizzare: quella corretta o quella nella quale abbiamo race conditions

La dinamica del rendering con CUDA

Analisi dei punti più importanti dell'implementazione – prima fase

Nella funzione `__global__ void kernelRenderCircles()` del file *cudaRenderer.cu* avviene il grosso della magia.

A ogni blocco di thread di dimensione `ThrInBlock_X` per `ThrInBlock_Y` è associata una porzione quadrata dell'immagine delle stesse dimensioni:

```
short blockLeftIdx =
blockIdx.x * ThrInBlock_X;
short blockRightIdx =
blockLeftIdx + ThrInBlock_X - 1;
short blockTopIdx =
blockIdx.y * ThrInBlock_Y;
short blockBottomIdx =
blockTopIdx + ThrInBlock_Y - 1;
```

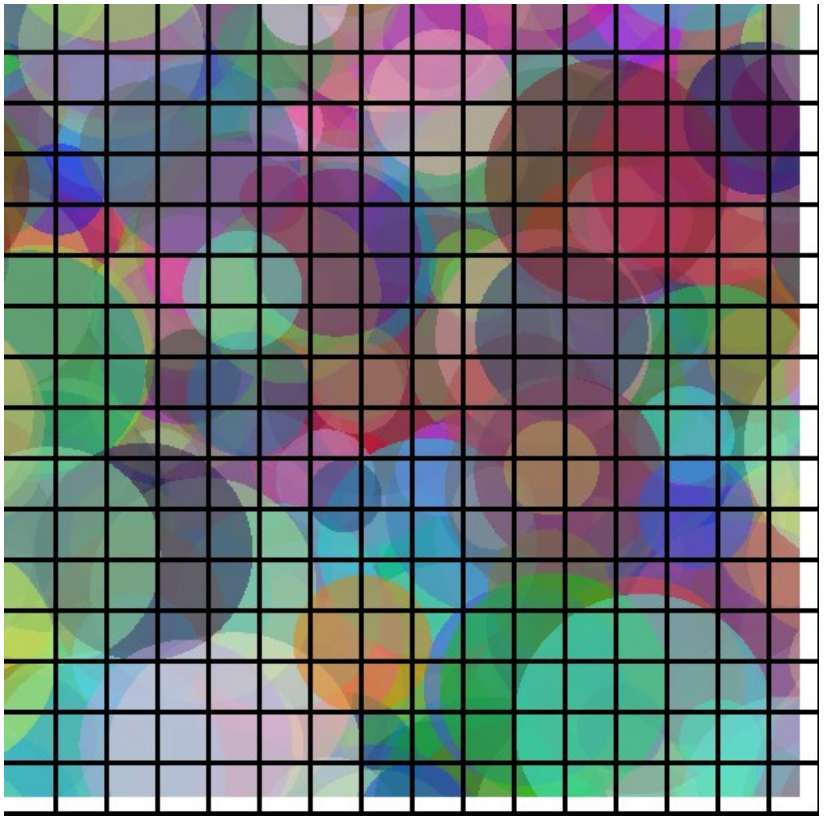
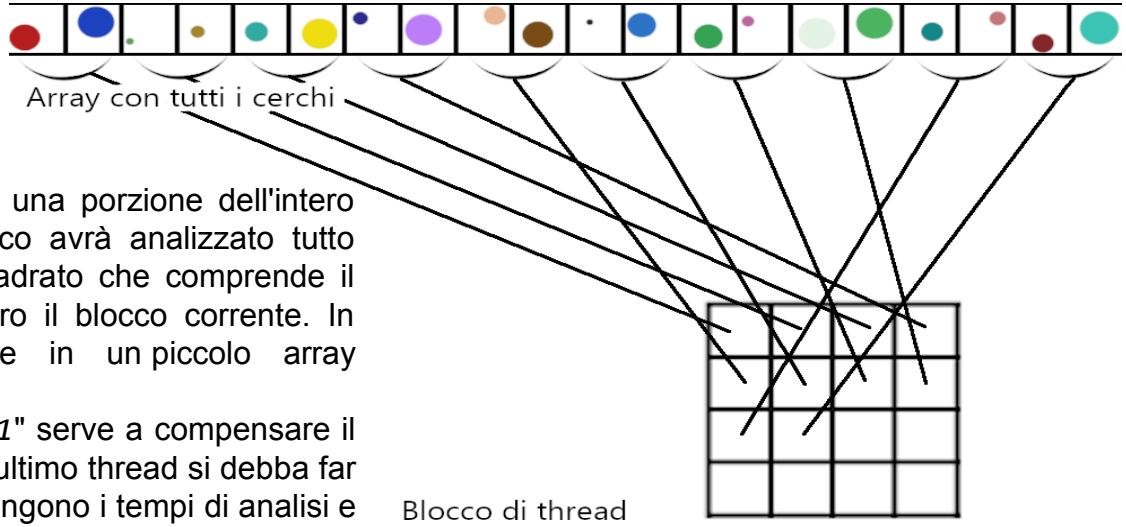


Immagine finale suddivisa in blocchi

La dinamica del rendering con CUDA

Analisi dei punti più importanti dell'implementazione – prima fase



Ogni thread in un blocco analizza una porzione dell'intero array dei cerchi (quindi ogni blocco avrà analizzato tutto l'array) al fine di valutare se il quadrato che comprende il cerchio ha almeno un punto dentro il blocco corrente. In caso positivo ne memorizza l'indice in un piccolo array locale.
Il "*numCircles + ThrInBlock - 1*" serve a compensare il troncamento a int, e a evitare che l'ultimo thread si debba far carico di troppi cerchi. Così si contengono i tempi di analisi e non si sprecano, per gli altri thread, cicli a vuoto.

```
int circlesPerThread = (cuConstRendererParams.numCircles + ThrInBlock - 1) / ThrInBlock;
int circleStartIdx = threadIndex * circlesPerThread;
int circleEndIdx = circleStartIdx + circlesPerThread;
// l'ultimo thread si prende tutti i restanti cerchi
if (threadIndex == ThrInBlock) {
    circleEndIdx = cuConstRendererParams.numCircles;
}
```

La dinamica del rendering con CUDA

Analisi dei punti più importanti dell'implementazione – prima fase

```
int threadCircleCount = 0;
int threadCircleList[CirclesPerThread];
for (int c = circleStartIdx; c<circleEndIdx; c++){
    if (c >= cuConstRendererParams.numCircles)
        break;
    float3 position = *(float3*)(&cuConstRendererParams.position[c * 3]);
    float radius = cuConstRendererParams.radius[c];
    if (circleInBoxConservative(position.x, position.y, radius, blockLeftNormIdx,
        blockRightNormIdx, blockBottomNormIdx, blockTopNormIdx) == 1){
        threadCircleList[threadCircleCount++] = c;
    }
}
```

Controllo sempre di non sbordare dall'array di cerchi

Uso la funzione `circleInBoxConservative` fornita col materiale iniziale per controllare se il quadrato che comprende il cerchio ha almeno un punto dentro il blocco corrente, e nel caso mi salvo il suo indice in un piccolo array locale.

La dinamica del rendering con CUDA

Analisi dei punti più importanti dell'implementazione – seconda fase

Adesso `threadCircleCount` contiene, della parte dell'array di cerchi analizzati, quanti stanno nel blocco corrente, e in `threadCircleList`, della parte dell'array di cerchi analizzati, gli indici di tali cerchi. Quindi questi risultati parziali vengono tutti caricati in un array condiviso a livello di blocco.

```
__shared__ uint circleCount[ThrInBlock];
__shared__ uint circleList[MaxCircles];
__shared__ uint indexList[ThrInBlock];

circleCount[threadIndex] = threadCircleCount;
__syncthreads();
sharedMemExclusiveScan(threadIndex, circleCount, indexList, circleList, ThrInBlock);
__syncthreads();
uint privateIndex = indexList[threadIndex];
for(int i=0; i<threadCircleCount; i++){
    circleList[privateIndex++] = threadCircleList[i];
}
__syncthreads();
```

`circleCount` è quindi una successione che contiene per ogni thread il numero di cerchi (appartenenti al blocco corrente) trovati nel rispettivo segmentino dell'array di tutti i cerchi.

Adesso, utilizzo la funzione `sharedMemExclusiveScan` fornita col materiale iniziale per calcolare la serie della successione `circleCount` e la metto in `indexList`, così da permettere poi ai thread di avere il corretto indice di partenza (`privateIndex`) a partire da cui copiare in `circleList` gli indici dei cerchi relativi al loro segmentino.

La dinamica del rendering con CUDA

Analisi dei punti più importanti dell'implementazione – terza fase

La serie non contiene l'ultima somma (è esclusiva).
La aggiungo ora per ottenere il totale dei cerchi nel blocco.

```
int totalCircles = indexList[ThrInBlock-1] + circleCount[ThrInBlock-1];
```

Adesso che abbiamo la lista dei cerchi relativi al blocco corrente possiamo iniziare il rendering vero e proprio.
Calcolo la coordinata continua del centro del pixel, campionando poi il colore del pixel sulla base del fatto che il cerchio includa effettivamente il centro del pixel o meno, ovvero faccio ciò che in gergo CGI è definito Point-Sampling.

```
int pixelXCoord = blockLeftIdx + threadIdx.x;
int pixelYCoord = blockTopIdx + threadIdx.y;
float2 pixelCenterNorm = make_float2(invWidth * (static_cast<float>(pixelXCoord) + 0.5f),
    invHeight * (static_cast<float>(pixelYCoord) + 0.5f));

float4* imgPtr = (float4*)(&cuConstRendererParams.imageData[4 * (pixelYCoord * imageWidth +
pixelXCoord)]);
float4 pixelData = *imgPtr;
```

La dinamica del rendering con CUDA

Analisi dei punti più importanti dell'implementazione – terza fase

```
for (int i=0; i<totalCircles; i++){

    int circleIndex = circleList[i];
    float3 position = *(float3*)(&cuConstRendererParams.position[circleIndex * 3]);
    float radius = cuConstRendererParams.radius[circleIndex];
    float diffX = position.x - pixelCenterNorm.x;    float diffY = position.y - pixelCenterNorm.y;
    float pixelDist = diffX * diffX + diffY * diffY; float maxDist = radius * radius;

    if (pixelDist <= maxDist){
        int index3 = 3 * circleIndex;
        float3 rgb = *(float3*)&(cuConstRendererParams.color[index3]);
        float alpha = .5f;
        float oneMinusAlpha = 1.f - alpha;
        pixelData.x = alpha * rgb.x + oneMinusAlpha * pixelData.x;
        pixelData.y = alpha * rgb.y + oneMinusAlpha * pixelData.y;
        pixelData.z = alpha * rgb.z + oneMinusAlpha * pixelData.z;
        pixelData.w = alpha + pixelData.w;
    }
}
*imgPtr = pixelData;
```

Adesso ogni thread renderizza un pixel ciclando su tutti i cerchi del blocco, partendo dal cerchio più lontano.

Questa invariante è data dal fatto che i cerchi vengono già dati in ordine di lontananza decrescente e sono mantenuti in questo ordine nella copia degli indici in `circleList`.

Controllo se il cerchio è effettivamente sul pixel o se lo era solo il quadrato che lo contiene.

Scrivo infine il colore risultante nell'immagine finale.

La dinamica del rendering con CUDA

Altri aspetti importanti dell'algoritmo

Al fine di ridurre al minimo gli accessi alla global memory, tutti i dati immutabili sono stati copiati in constant memory:

```
GlobalConstants params;
params.sceneName = sceneName;
params.numCircles = numCircles;
params.imageWidth = image->width;
params.imageHeight = image->height;
params.position = cudaDevicePosition;
params.color = cudaDeviceColor;
params.radius = cudaDeviceRadius;
params.imageData = cudaDeviceImageData;

cudaMemcpyToSymbol(cuConstRendererParams, &params, sizeof(GlobalConstants));
```

L'algoritmo utilizza la global memory solo per leggere la matrice di pixel e per sovrascriverla.

Analisi prestazionale del rendering

Confronto tra i due diversi approcci al rendering

Sfruttando la funzione integrata di benchmark, la quale produce per 5 volte i rendering tramite CPU e GPU degli stessi cerchi, e li confronta, possiamo analizzare lo speedup al variare di alcuni parametri, tra cui numero di cerchi, risoluzione dell'immagine finale, e dimensione dei blocchi di thread.

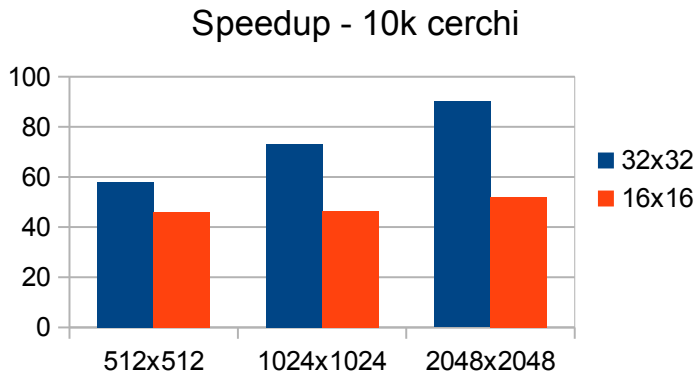
I seguenti benchmark sono eseguiti su:
Laptop Xiaomi Mi Notebook Pro
CPU Intel Core i7-8550U (8 core logici a 1.8 GHz - fino a 4 GHz in Turbo Boost)
RAM 16 GB DDR4
S.O. Manjaro Linux 18.1.4 (basato su Arch Linux) con Kernel Linux 5.4
Chip grafico Nvidia GeForce MX150
Streaming Multiprocessors: 3 Memoria Globale: 2003 MB
Compute Capability: 6.1 Shared memory per blocco: 48 KB
Librerie CUDA 10.1

Al fine di poter utilizzare il chip grafico solo su richiesta ho dovuto installare i pacchetti bumblebee e optirun che sono tentativi di portare la tecnologia NVidia Optimus in ambiente open source Linux. In quanto pacchetti non ufficiali NVidia, le prestazioni ottenute col chip grafico potrebbero non essere quelle massime.

Analisi prestazionale del rendering

Tempi medi di rendering per 10k cerchi – misure in millisecondi

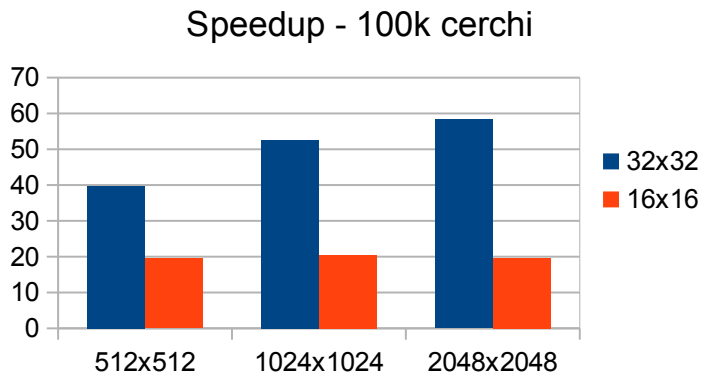
	Ris. 512x512 Blocchi 32x32	Ris. 512x512 Blocchi 16x16	Ris. 1024x1024 Blocchi 32x32	Ris. 1024x1024 Blocchi 16x16	Ris. 2048x2048 Blocchi 32x32	Ris. 2048x2048 Blocchi 16x16
CPU	488	490	1827	1840	7050	7046
GPU	8	11	25	40	78	135
Speedup	57,9	46,1	73,3	46,5	90,1	52,1



Analisi prestazionale del rendering

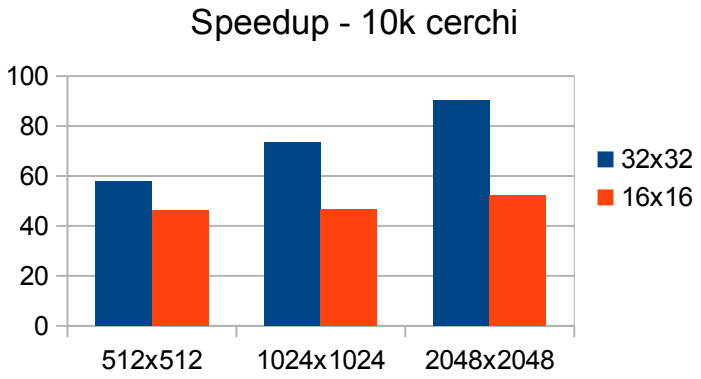
Tempi medi di rendering per 100k cerchi – misure in millisecondi

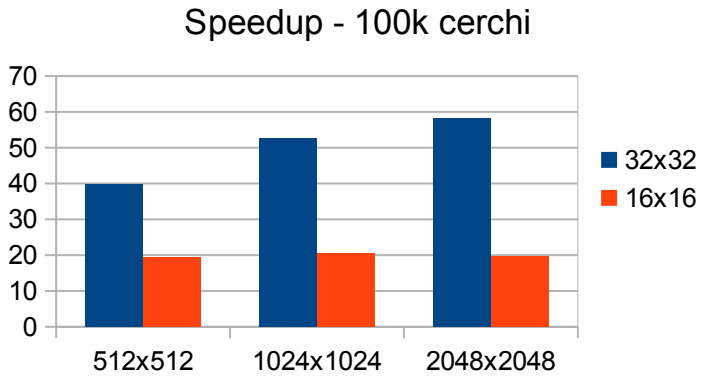
	Ris. 512x512 Blocchi 32x32	Ris. 512x512 Blocchi 16x16	Ris. 1024x1024 Blocchi 32x32	Ris. 1024x1024 Blocchi 16x16	Ris. 2048x2048 Blocchi 32x32	Ris. 2048x2048 Blocchi 16x16
CPU	4464	4426	17812	17819	69951	69906
GPU	112	226	338	872	1198	3544
Speedup	39,7	19,5	52,6	20,4	58,3	19,7



Analisi prestazionale del rendering

Considerazioni finali





Mettendo a confronto i due grafici dello speedup ci rendiamo immediatamente conto del fatto che con blocchi di 32x32 thread abbiamo prestazioni GPU molto migliori rispetto a blocchi di 16x16. Si può inoltre notare che, nel caso di blocchi 32x32 lo speedup tende ad aumentare all'aumentare della risoluzione, indipendentemente dal numero di cerchi renderizzati.
Nel caso con 10k cerchi con immagine a risoluzione 2048x2048 arriviamo a un fattore di speedup di addirittura 90 volte!
Ciò mette in luce il fatto che le GPU possono essere davvero vantaggiose per compiti quali il rendering di figure sullo schermo.

Andrea Spitaleri