Problemi di Render: Parallel Programming for Machine Learning

Lorenzo Baiardi, Thomas Del Moro

28 03 2024

1 Introduzione

In questo documento, analizzeremo l'efficacia dell'applicazione di diversi metodi di parallelizzazione sul problema di compositing tra piani, osservando in dettaglio i tempi di esecuzione associati. Il task consiste nel costruire un Renderer di cerchi di colore diverso e parzialmente trasparenti, attraverso i quali si possano quindi intravedere i cerchi dei piani sottostanti.

2 Analisi del problema

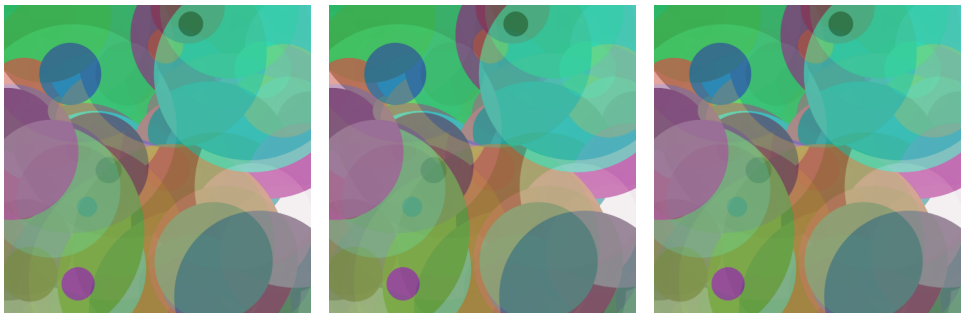
Introduciamo alcune definizioni che ci torneranno utili:

- n : numero di cerchi per ogni piano
- N : numero di piani da sovrapporre
- D : dimensione 2D di ogni piano

Ciascun piano è caratterizzato da quattro canali di colore (RGBA, ovvero RGB con trasparenza), in modo da creare un effetto visivo i trasparenza. Su ogni piano vengono disegnati n cerchi di colore diverso, dopodiché tali immagini vengono sovrapposte dalla prima all'ultima. In questa fase sarà utilizzato il canale di trasparenza e sarà importante l'ordine con cui verrà eseguito il compositing. L'immagine risultante sarà un insieme dei cerchi dell'ultimo piano, attraverso i quali si intravederanno quelli dei piani sottostanti.

Eseguiremo il task sia in maniera sequenziale che in modo parallelo, utilizzando diverse tecniche e linguaggi di parallelizzazione. In Figura 1 è

mostrato un esempio del risultato ottenuto con $N = 3000$ e $n = 50$ usando i diversi metodi di parallelizzazione testati.



(a) Sequenziale (b) OMP (c) CUDA

Figura 1: Immagini di esempio

3 Implementazione

Abbiamo realizzato il progetto in linguaggio C++, facendo uso della libreria OpenCV per la realizzazione delle immagini. Innanzitutto vengono generati casualmente i cerchi e inseriti nei rispettivi piani. Tutti tali dati vengono salvati in un array di matrici di OpenCV che poi saranno processati.

```
1  Circle* sequentialGenerateCircles(std::size_t n, int width, int height
       , int minRadius, int maxRadius) {
2      auto* circles = new Circle[n];
3      std::mt19937 generator(std::random_device{}());
4
5      std::uniform_int_distribution<int> colorDistribution(0, 255);
6      std::uniform_int_distribution<int> pointXDistribution(1, width);
7      std::uniform_int_distribution<int> pointYDistribution(1, height);
8      std::uniform_int_distribution<int> radiusDistribution(minRadius,
       maxRadius);
9
10     for (int i = 0; i < n; i++) {
11         cv::Scalar color(colorDistribution(generator),
       colorDistribution(generator), colorDistribution(generator), 255);
12         cv::Point center(pointXDistribution(generator),
       pointYDistribution(generator));
13         int r = radiusDistribution(generator);
14         circles[i] = Circle{color, center, r};
15     }
16     return circles;
17 }
18
19 cv::Mat* sequentialGeneratePlanes(std::size_t nPlanes, Circle circles
       [], std::size_t nCircles) {
```

```
    auto *planes = new cv::Mat[nPlanes];

    for (int i = 0; i < nPlanes; i++) {
        planes[i] = TRANSPARENT_MAT;
        for (int j = 0; j < nCircles; j++) {
            auto circle = circles[i * nCircles + j];
            cv::circle(planes[i], circle.center, circle.r, circle.color, cv::FILLED, cv::LINE_AA);
        }
    }
    return planes;
}
```

Una volta generati i piani separatamente, questi vengono sovrapposti sequenzialmente l'uno con l'altro, sommando a ogni iterazione il piano corrente al risultato dei piani precedenti.

```
double sequentialRenderer(cv::Mat planes[], std::size_t nPlanes) {
    cv::Mat result = TRANSPARENT_MAT;
    int cn = result.channels();

    // START
    double start = omp_get_wtime();

    for (int i = 0; i < result.rows; i++) {
        for (int j = 0; j < result.cols; j++) {
            for (int z = 0; z < nPlanes; z++) {
                cv::Mat *src = &planes[z];
                for (int c = 0; c < cn; c++)
                    result.data[i * result.step + cn * j + c] =
                            result.data[i * result.step + j * cn + c] * (1 - ALPHA) +
                            src->data[i * src->step + j * cn + c] * ALPHA);
            }
        }
    }

    double time = omp_get_wtime() - start;
    // END

    cv::imwrite(SEQ_IMG_PATH + std::to_string(nPlanes) + ".png", result);
    return time;
}
```

Vediamo dunque in modo più specifico le tecniche di parallelizzazione utilizzate.

4 Parallelizzazione

Per parallelizzare la problematica, abbiamo optato per l'adozione di due approcci distinti. Nel primo scenario, abbiamo impiegato la libreria OpenMP,

che permette di eseguire più thread contemporaneamente sulla CPU della nostra macchina. Nel secondo caso, abbiamo invece sfruttato il linguaggio di programmazione CUDA per eseguire calcoli in parallelo sulla GPU a disposizione.

4.1 OpenMP

OpenMP si tratta di un API per la programmazione parallela su sistemi a memoria condivisa. È composto da un insieme di direttive di compilazione, routine di librerie e variabili d'ambiente che consentono di sviluppare applicazioni di calcolo parallele sfruttando i diversi core delle CPU moderne.

4.1.1 Generazione dei cerchi

Poiché la generazione dei cerchi impiega tempi molto lunghi, abbiamo prima di tutto utilizzato OpenMP per velocizzare questo processo.

```
1  Circle* parallelGenerateCircles(std::size_t n, int width, int height,
       int minRadius, int maxRadius) {
2      auto* circles = new Circle[n];
3      std::mt19937 generator(std::random_device{}());
4      std::uniform_int_distribution<int> colorDistribution(0, 255);
5      std::uniform_int_distribution<int> pointXDistribution(1, width);
6      std::uniform_int_distribution<int> pointYDistribution(1, height);
7      std::uniform_int_distribution<int> radiusDistribution(minRadius,
       maxRadius);
8
9  #pragma omp parallel for default(none) shared(circles, generator)
10     for (int i = 0; i < n; i++) {
11         cv::Scalar color(colorDistribution(generator),
       colorDistribution(generator), colorDistribution(generator), 255);
12         cv::Point center(pointXDistribution(generator),
       pointYDistribution(generator));
13         int r = radiusDistribution(generator);
14         circles[i] = Circle{color, center, r};
15     }
16     return circles;
17 }
18
19 cv::Mat* parallelGeneratePlanes(std::size_t nPlanes, Circle circles[],
        std::size_t nCircles) {
20     auto *planes = new cv::Mat[nPlanes];
21
22 #pragma omp parallel for default(none) shared(planes, circles)
       firstprivate(nPlanes, nCircles)
23     for (int i = 0; i < nPlanes; i++) {
24         planes[i] = TRANSPARENT_MAT;
25         for (int j = 0; j < nCircles; j++) {
26             auto circle = circles[i * nCircles + j];
```

```
27              cv::circle(planes[i], circle.center, circle.r, circle.
    color, cv::FILLED, cv::LINE_AA);
28          }
29      }
30      return planes;
31  }
```

Come si può notare nello codice sopra, alla riga 10 è stata aggiunta una direttiva per la definizione di un loop parallelo, eseguito quindi da più thread contemporaneamente. Poiché l'ordine in cui i cerchi vengono generati non è importante, si tratta di un problema imbarazzantemente parallelizzabile. Allo stesso modo, alla riga 24 è stato definito un secondo loop parallelo per suddividere l'assegnazione tra cerchi e piani tra thread diversi.

4.1.2 Rendering

Passiamo adesso ad analizzare il task principale, ovvero quello del compositing dei piani. Dopo aver esplorato varie possibilità, l'approccio migliore è risultato quello di assegnare a ciascun thread la sommatoria su tutti i piani di un singolo pixel per ogni matrice. Questo metodo permette di mantenere l'ordine dei piani, poiché per ogni pixel la sommatoria viene eseguita sequenzialmente, ma allo stesso tempo incrementa la velocità di rendering complessiva dato che pixel diversi sono assegnati a thread diversi ed elaborati contemporaneamente. Un thread, dunque, a ogni iterazione utilizza il pixel risultante dai piani precedenti e somma a questo il pixel del piano corrente. Una volta completata l'operazione per tutti i piani il thread viene assegnato a un nuovo pixel finché la matrice 2D non si è esaurita.

```
1   double parallelRenderer(cv::Mat planes[], std::size_t nPlanes) {
2       cv::Mat result = TRANSPARENT_MAT;
3       int cn = result.channels();
4
5       // START
6       double start = omp_get_wtime();
7
8   #pragma omp parallel for default(none) shared(result, planes)
        firstprivate(nPlanes, cn) collapse(2)
9       for (int i = 0; i < result.rows; i++) {
10          for (int j = 0; j < result.cols; j++) {
11              for (int z = 0; z < nPlanes; z++) {
12                  cv::Mat *src = &planes[z];
13                  for (int c = 0; c < cn; c++)
14                      result.data[i * result.step + cn * j + c] =
15                              result.data[i * result.step + j * cn + c]
        * (1 - ALPHA) +
16                              src->data[i * src->step + j * cn + c] * (
        ALPHA);
17              }
18          }
```

```
19      }
20
21      double time = omp_get_wtime() - start;
22      // END
23
24      cv::imwrite(PAR_IMG_PATH + std::to_string(nPlanes) + ".png",
        result);
25      return time;
26  }
```

Come si può notare dallo snippet di codice sopra, alla riga 8 è stata inserita una direttiva OpenMP per definire un loop parallelo. La clausola *collapse*(2) permette di parallelizzare sui primi due cicli più esterni, ovvero sia sulle righe che sulle colonne della matrice.

4.2 CUDA

CUDA è un architettura hardware per l'elaborazione parallela sviluppata da NVIDIA, che permette di sviluppare codice eseguibile su GPU. La GPU permette di fare affidamento su un gran numero di core a differenza del singolo processore e ciò permette di realizzare applicazioni parallele estremamente veloci.

Anche in questo caso abbiamo valutato diversi approcci, di cui alcuni sono risultati più efficienti di altri. In particolare, il metodo che sembra ottenere il miglior speedup è mostrato nel seguente snippet di codice, in cui si affida a ogni thread la sommatoria su tutti i piani di un solo pixel, analogamente a quanto fatto con OpenMP. Costruiamo blocchi 8x8, 16x16 0 32x32 in modo da non superare i limite della nostra GPU di 1024 thread per blocco ma allo stesso tempo da avere sempre un numero di thread multiplo di 32 per ottenere un vantaggio nell'istanziamento degli warp. Il numero di blocchi varia quindi in base alla dimensione di ogni blocco, in modo da assegnare ciascun thread un pixel della matrice.

```
1       // GRID AND BLOCK DIMENSIONS
2       dim3 block(blockSize, blockSize); // threads x block
3       dim3 grid(result.cols / block.x, result.rows/ block.y); // blocks
```

```
1
2   __global__ void cudaKernelCombinePlanes(uchar4* resultData, const
        uchar4* planesData,
3                                           int width, int height, int
        nPlanes) {
4       auto x = blockIdx.x * blockDim.x + threadIdx.x;
5       auto y = blockIdx.y * blockDim.y + threadIdx.y;
```

```
6
7       if (x < width && y < height) {
8           auto idx = y * width + x;
9           auto oneMinusAlpha = 1.0f - ALPHA;
10          auto result = resultData[idx];
11
12          for (int z = 0; z < nPlanes; z++) {
13              auto idxP = z * width * height + idx;
14              const auto &plane = planesData[idxP];
15
16              result.x = result.x * oneMinusAlpha + plane.x * ALPHA;
17              result.y = result.y * oneMinusAlpha + plane.y * ALPHA;
18              result.z = result.z * oneMinusAlpha + plane.z * ALPHA;
19              result.w = result.w * oneMinusAlpha + plane.w * ALPHA;
20          }
21          resultData[idx] = result;
```

Notiamo che in questo modo la sommatoria sui piani viene eseguita in modo sequenziale da un singolo thread, dunque l'immagine risultate non subisce trasformazioni errate.

5 Caratteristiche della macchina

Per condurre i test con OpenMP e CUDA, abbiamo utilizzato due macchine diverse:
La macchina utilizzata per effettuare i test con OpenMP è dotata di:

- **CPU**: Intel Core i7-1360P (4 P-Core, 8 E-Core, 12 Cores, 16 Threads)
- **RAM**: 16 GB
- **Sistema Operativo**: Windows 11 Home
- **Librerie**: OpenCV 4.6.0

La macchina utilizzata per effettuare i test con CUDA è dotata di:

- **CPU**: Intel Core i5-8600K 3.60 GHz (6 core)
- **RAM**: 16 GB
- **GPU** NVIDIA GeForce GTX 1050 Ti 4 GB
- **Sistema Operativo**: Windows 11 PRO
- **Librerie**: OpenCV 4.6.0, CUDA 11.8

6 Esperimenti e Risultati

Gli esperimenti sono stati condotti variando principalmente il numero di piani da combinare e la dimensione delle immagini. Prima dell'esecuzione dei test, sono stati generati tutti i piani e i cerchi in modo da garantire un confronto accurato tra i diversi metodi di parallelizzazione.
Le dimensioni delle immagini utilizzate sono 256x256, 512x512 e 1024x1024. Per gli esperimenti su OpenMP, il numero di piani è stato variato da 100 a un massimo di 1000, con un incremento di 100 piani. Nell'appendice B sono riportati i risultati ottenuti con un massimo di 10000 piani. Per gli esperimenti su CUDA invece, il numero di piani è stato variato da 1000 a un massimo di 10000, con un incremento di 1000 piani per i test con risoluzioni 256x256 e 512x512. Per il test con risoluzione 1024x1024, il numero di piani varia in un range più piccolo.

6.1 OpenMP

In questa sezione analizziamo gli esperimenti eseguiti con la versione sequenziale e quella parallelizzata con OpenMP.

6.1.1 Generazione dei cerchi

Prima di tutto ci siamo occupati di osservare i risultati ottenuti con la parallelizzazione della generazione dei piani e dei cerchi. Nella Tabella 1 sono riportati i tempi impiegati per la generazione su immagini di dimensioni 512x512 utilizzando 16 thread. Inoltre nelle Tabelle 3, 4 riportate nell'appendice sono presenti gli stessi risultati ottenuti su immagini di dimensioni diverse.

	$N = 100$		$N = 1000$		$N = 10000$	
	$n = 50$	$n = 500$	$n = 50$	$n = 500$	$n = 50$	$n = 500$
Sequential (s)	0.349	1.404	3.332	14.071	33.372	151.407
Parallel (s)	0.045	0.180	0.326	1.701	3.257	17.310
Speedup	7.756	7.800	10.221	8.272	**10.246**	**8.747**

Tabella 1: Tempi impiegati per la generazione dei cerchi e dei piani su immagini 512x512

Dalle tabelle risulta evidente che si abbia un enorme vantaggio nell'utilizzo di OpenMP rispetto alla versione sequenziale. Ciò è dovuto al fatto che la generazione dei piani e dei cerchi è un'operazione molto intensiva in

termini di calcolo, e quindi la parallelizzazione porta a un notevole miglioramento delle performance. La parallelizzazione della generazione dei cerchi ha anche permesso di eseguire più velocemente i test seguenti.

6.1.2 Rendering

Per quanto riguarda l'operazione principale, ovvero il compositing dei piani, i risultati sono stati suddivisi per dimensione delle immagini. Per ognuna di queste dimensioni è stato variato il numero di piani da combinare.
Le immagini 256x256 sono le più piccole testate. In Figura 2 sono riportati i tempi di esecuzione (sulla sinistra) e i corrispondenti speedup (sulla destra) per le diverse combinazioni di piani e cerchi.

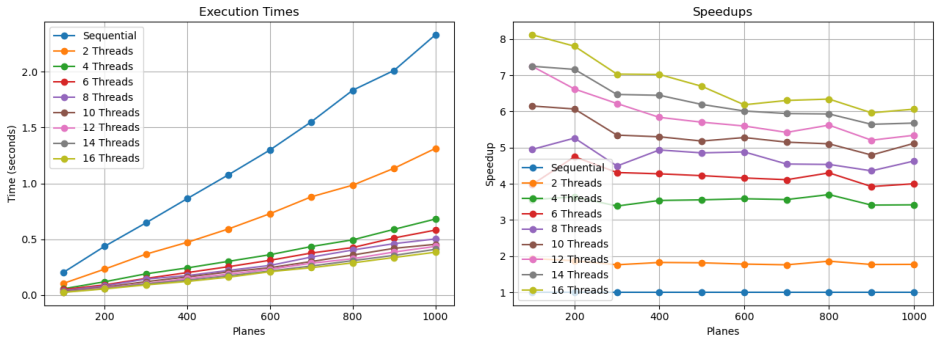


Figura 2: Tempi di esecuzione e speedup con OpenMP su immagini 256x256

Appare evidente che maggiore è il numero di thread utilizzati, minore è il tempo di esecuzione. Con 16 thread infatti si ottiene il migliore speedup che, a eccezione delle prime quantità di piani, rimane abbastanza costante per tutto l'esperimento.

Dopodiché abbiamo eseguito gli stessi test su immagini di dimensioni maggiori, ovvero 512x512. I risultati in Figura 3, mostrano che con immagini di dimensione maggiore, il valore dello speedup cresce leggermente. D'altra parte possiamo notare che un numero di thread molto alto porta a una riduzione delle performance, soprattutto quando i piani diventano tanti.

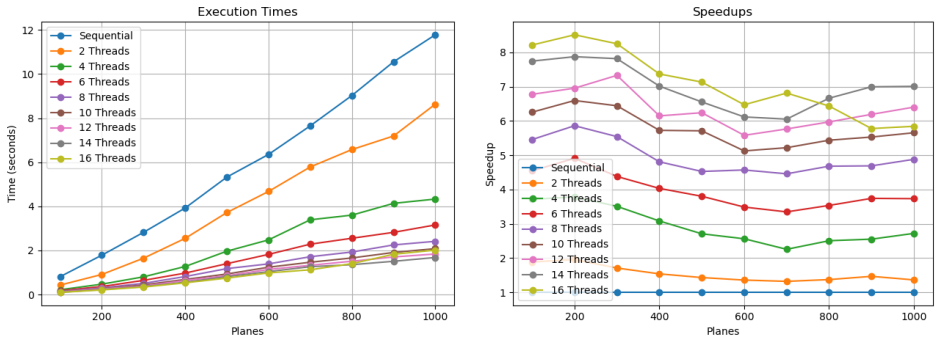


Figura 3: Tempi di esecuzione e speedup con OpenMP su immagini 512x512

Infine abbiamo eseguito gli stessi esperimenti su immagini 1024x1024. In Figura 4 possiamo osservare che, con immagini così grandi, i tempi di esecuzione diventano molto maggiori. Lo speedup però sembra non cambiare molto rispetto ai test su immagini più piccole, probabilmente a causa di un'eccessiva memoria occupata da tali piani.

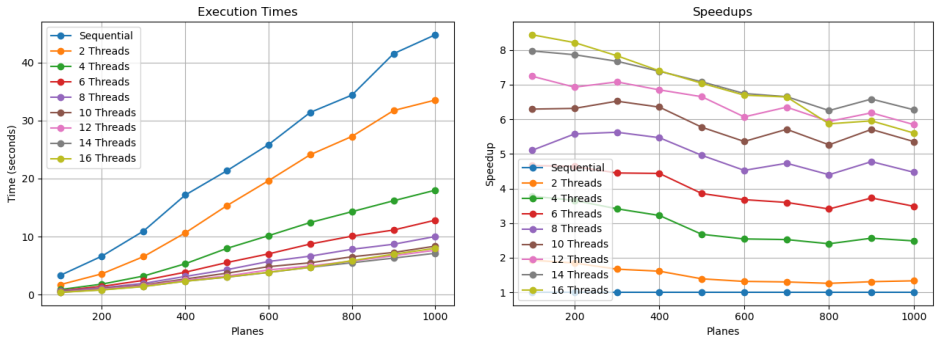


Figura 4: Tempi di esecuzione e speedup con OpenMP su immagini 1024x1024

In tutti i casi lo speedup è rimasto abbastanza costante. In generale con un numero maggiore di thread si sono ottenuti risultati migliori, ma un numero molto alto di thread affiancato a una grande quantità di memoria utilizzata, ha determinato un peggioramento delle performance.

6.2 CUDA

Passiamo adesso ad analizzare gli esperimenti eseguiti utilizzando CUDA come metodo di parallelizzazione. Ovviamente ci aspettiamo valori di speedup molto maggiori dei precedenti, dato l'utilizzo della GPU a disposizione.

6.2.1 Dimensione dei thread-blocks

Come abbiamo visto, per la costruzione della griglia e dei blocchi della GPU è necessario scegliere una dimensione appropriata. Abbiamo quindi eseguito degli esperimenti per determinare la dimensione ottimale dei thread-blocks. In particolare, sappiamo che conviene utilizzare un numero di thread per blocco multiplo di 32 e relativamente grande, in modo da sfruttare al meglio le istanze degli warp. Allo stesso tempo la nostra GPU permette un massimo di 1024 thread per blocco. Abbiamo quindi testato le performance con blocchi di dimensione 8x8, 16x16 e 32x32. I risultati riportati in Tabella 2 mostrano i tempi di esecuzione dell'operazione di rendering al variare della dimensione delle immagini e del numero di piani.

	Dimensione dell'immagine					
	$256x256$		$512x512$		$1024x1024$	
	$N = 500$	$N = 5000$	$N = 500$	$N = 5000$	$N = 500$	$N = 2000$
$8x8$	0.024	0.189	0.086	0.884	0.252	2.507
$16x16$	**0.021**	**0.157**	0.073	**0.645**	**0.249**	1.214
$32x32$	0.022	0.161	**0.066**	0.657	**0.249**	**1.107**

Tabella 2: Tempi (secondi) impiegati per il rendering con diverse dimensioni dei thread-blocks di CUDA

Come ci aspettavamo, i blocchi 8x8 sono risultati in tutti i casi quelli meno performanti. I blocchi 16x16 sembrano fare leggermente meglio di quelli 32x32, anche se entrambi determinano tempi di esecuzione simili. Dunque per tutti i prossimi esperimenti abbiamo utilizzato blocchi 16x16, ovvero da 256 threads.

6.2.2 Rendering

Poiché abbiamo notato che la dimensione ottimale dei thread-blocks è 32x32, abbiamo eseguito tutti i prossimi test seguendo questa configurazione. Anche in questo caso gli esperimenti sono suddivisi sulla base della dimensione delle immagini.

Come nel caso di OpenMP, il primo test è stato eseguito su immagini 256x256 e il numero di piani è stato variato tra un minimo di 1000 e un massimo di 10000. I risultati sono riportati in Figura 5.

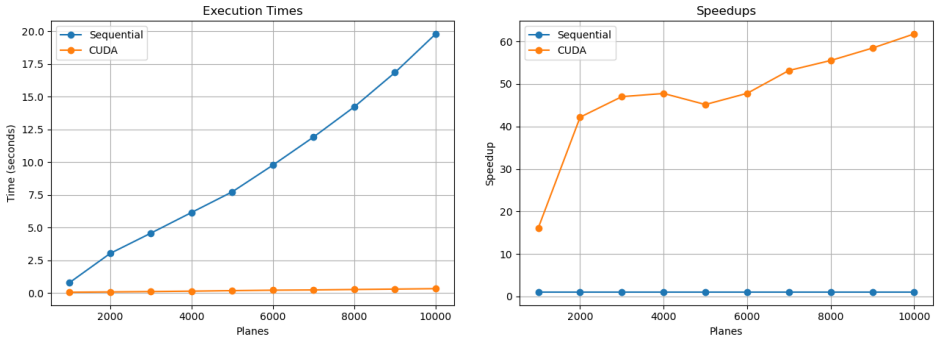


Figura 5: Tempi di esecuzione e speedup con CUDA su immagini 256x256

Come ci aspettavamo, il tempo di esecuzione è molto minore rispetto a quello ottenuto eseguendo le operazioni sulla CPU. Lo speedup aumenta man mano che cresce il numero di piani combinati e si ottengono tempi circa 60 volte minori rispetto alla versione sequenziale quando i piani arrivano intorno al loro massimo.

Di seguito invece abbiamo riportati i risultati degli stessi esperimenti eseguiti su immagini 512x512.

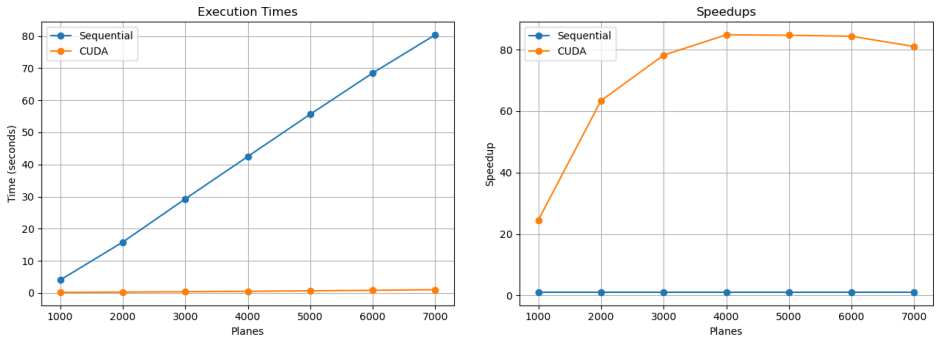


Figura 6: Tempi di esecuzione e speedup con CUDA su immagini 512x512

Essendo aumentata la quantità di pixel da elaborare, lo sfruttamento della GPU è diventato ancora più vantaggioso. I tempi impiegati per il

compositing sono ovviamente più lunghi dei precedenti, specialmente per la versione sequenziale, dunque anche lo speedup migliora rispetto alle immagini più piccole e tocca un massimo di 85 intorno ai 4000-6000 piani.

Infine abbiamo eseguito gli stessi esperimenti su immagini 1024x1024, i cui risultati sono mostrati in Figura 7. In questo caso, a causa di limitazioni computazionali della GPU a disposizione, abbiamo potuto eseguire i test sono con un numero minore di piani. I risultati che abbiamo ottenuto, fino al massimo di valori testati, rimangono coerenti con quelli ottenuti in precedenza. Ci possiamo ovviamente aspettare che con un numero di pixel ancora maggiore, lo speedup crescerebbe maggiormente rispetto ai casi sopra.

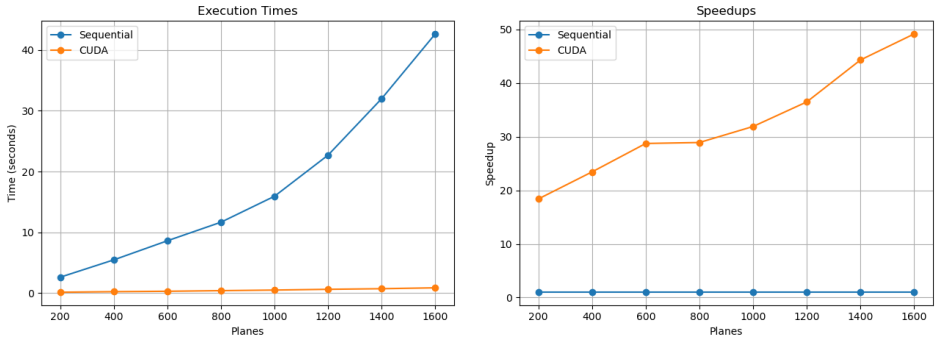


Figura 7: Tempi di esecuzione e speedup con CUDA su immagini 1024x1024

6.2.3 Utilizzo dei burst di memoria (CUDA-Color)

Poiché potrebbe essere vantaggioso sfruttare i burst di memoria per migliorare le performance, abbiamo eseguito degli esperimenti per valutare l'efficacia di questa tecnica nel caso del Rendering. A differenza del metodo precedente, abbiamo utilizzato un kernel diverso. L'array dei piani passato in ingresso è stato riordinato in modo da avere ogni canale di ogni pixel separato. In questo modo tale canale, gestito da un solo thread, sarà rappresentato dal suo valore su tutti i piani in maniera contigua. Ciò dovrebbe permettere di sfruttare il caricamento in burst dalla memoria globale della GPU.

```
1 }
2
3 __global__ void cudaKernelCombinePlanesColor(uchar4* d_resultData,
      const uchar* d_planesData,
```

```
                                                const int width, const
    int height, const int nPlanes) {

    auto x = blockIdx.x * blockDim.x + threadIdx.x;
    auto y = blockIdx.y * blockDim.y + threadIdx.y;

    if (x < width && y < height) {
        auto oneMinusAlpha = 1.0f - ALPHA;
        int channels = 4;
        auto idx = y * width * channels * nPlanes + x * channels *
    nPlanes;
        auto idxP = y * width + x;
        uchar threadData[] = {d_resultData[idxP].x, d_resultData[idxP
    ].y, d_resultData[idxP].z, d_resultData[idxP].w};

        for (int c = 0; c < channels; c++){
            for (int z = 0; z < nPlanes; z++) {
                threadData[c] = threadData[c] * oneMinusAlpha +
    d_planesData[idx + c * nPlanes + z] * ALPHA;
            }
        }
        d_resultData[idxP] = {threadData[0], threadData[1], threadData
    [2], threadData[3]};
```

Come è possibile osservare nello snippet di codice sopra, in questo caso l'array contenente le informazioni dei piani sarà di tipo *uchar* invece che *uchar*4 e ovviamente il codice del kernel è leggermente diverso. Si può notare che nel ciclo *for* viene sfruttata la memoria contigua per caricare i valori dei piani in un registro, in modo da poterli utilizzare per il calcolo del colore del pixel. I risultati ottenuti sono riportati in Figura 8.

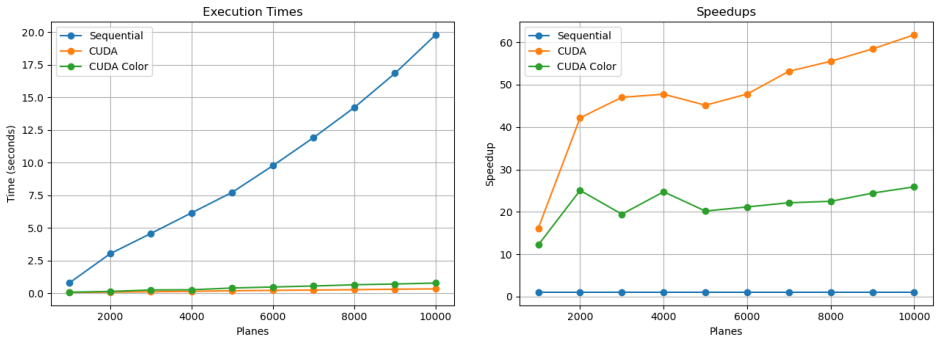


Figura 8: Tempi di esecuzione e speedup con CUDA-Color su immagini 256x256

A differenza di ciò che ci aspettavamo, i risultati sono a favore della versione che abbiamo visto finora. Ciò può dipendere dal fatto che in questo caso ogni thread legge un elemento *uchar* per volta e si trova quindi a ese-

guire il quadruplo delle letture dalla memoria globale rispetto alla versione precedente.

6.2.4 Copia dei dati in GPU

In tutti gli esperimenti che abbiamo eseguito, non abbiamo tenuto conto del tempo impiegato per la copia dei dati dalla CPU alla scheda grafica e viceversa. Questo è un aspetto molto importante da considerare, in quanto potrebbe influenzare notevolmente le performance. In Figura 9 sono mostrate le differenze tra i tempi di esecuzioni visti finora e quelli complessivi della copia dei dati da e verso la GPU.

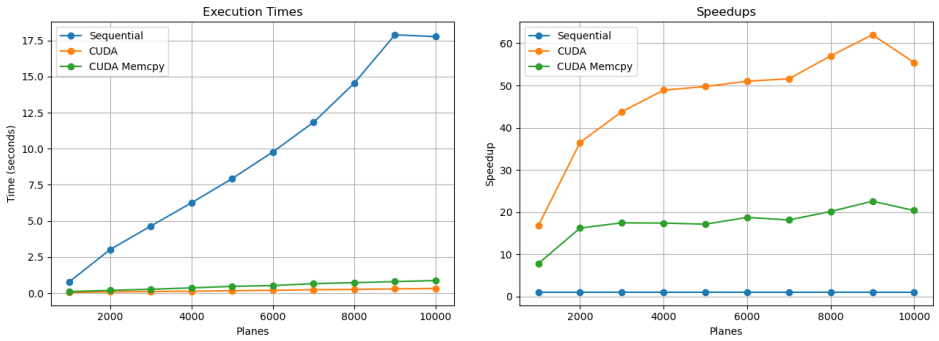


Figura 9: Tempi di esecuzione e speedup su immagini 256x256

Come ci aspettavamo, se consideriamo il tempo impiegato per il passaggio dei dati, i tempi si allungano e lo speedup cala significativamente. Ciò è indubbiamente dovuto al fatto che la quantità di dati scambiati tra l'host e la GPU è molto pesante e su di essa vengono poi eseguite operazioni relativamente leggere.

7 Conclusioni

In sintesi, come evidenziato dai test condotti, la parallelizzazione permette in ogni caso di gestire il problema di compositing dei piani in modo molto più veloce rispetto alla semplice computazione sequenziale. In particolare, l'approccio CUDA si è rivelato molto efficace per questo problema, permettendo di ottenere uno speedup molto significativo. OpenMP, pur non raggiungendo lo stesso livello di efficienza, ha comunque permesso di ottenere un miglioramento importante rispetto alla versione sequenziale. En-

trambi questi metodi diventano particolarmente utili quando la quantità di dati da processare aumenta molto, rendendo quasi impraticabile la soluzione sequenziale.

A Generazione dei cerchi

Nelle Tabelle seguenti sono mostrati i tempi impiegati per generare i piani e i cerchi al variare della dimensione dell'immagine.

	$N = 100$		$N = 1000$		$N = 10000$	
	$n = 50$	$n = 500$	$n = 50$	$n = 500$	$n = 50$	$n = 500$
Sequential	0.156	0.974	1.501	9.744	15.053	96.508
Parallel	0.029	0.148	0.166	1.172	1.627	11.558
Speedup	5.379	6.581	**9.042**	8.314	**9.252**	8.350

Tabella 3: Tempi impiegati per la generazione dei cerchi e dei piani su immagini 256x256

	$N = 100$		$N = 1000$		$N = 10000$	
	$n = 50$	$n = 500$	$n = 50$	$n = 500$	$n = 50$	$n = 500$
Sequential	1.067	2.625	11.061	23.820	108.643	273.588
Parallel	0.106	0.443	0.979	3.598	106.181	118.353
Speedup	10.066	5.926	**11.298**	6.620	1.023	2.312

Tabella 4: Tempi impiegati per la generazione dei cerchi e dei piani su immagini 1024x1024

B Rendering OpenMP

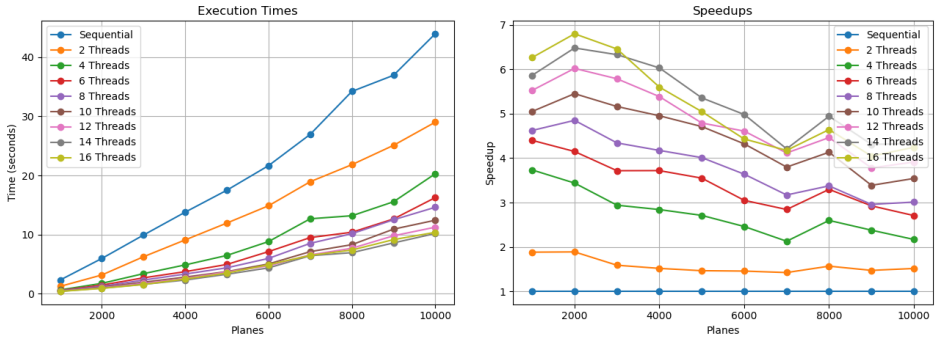


Figura 10: Tempi di esecuzione e speedup con OpenMP su immagini 256x256

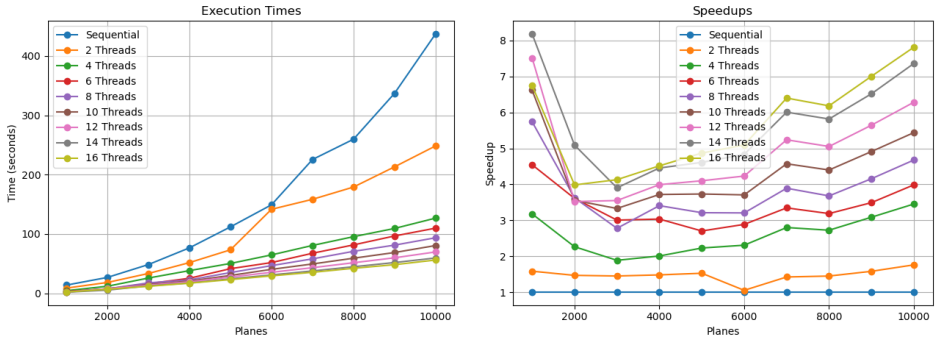


Figura 11: Tempi di esecuzione e speedup con OpenMP su immagini 512x512

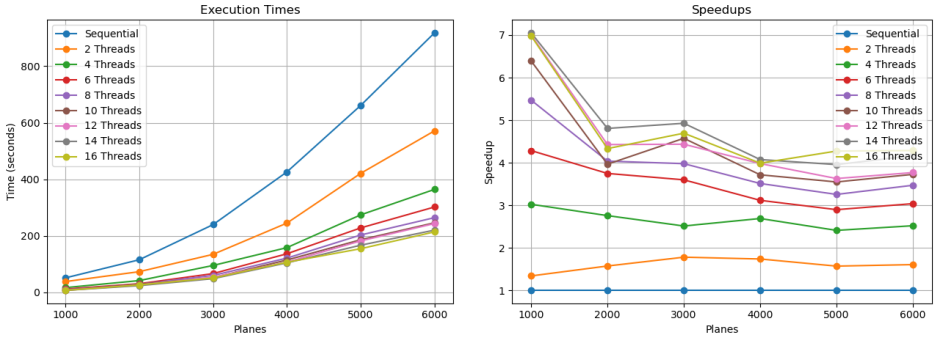


Figura 12: Tempi di esecuzione e speedup con OpenMP su immagini 1024x1024