Problemi di Render: Parallel Programming for Machine Learning

Lorenzo Baiardi, Thomas Del Moro

10 12 2023

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

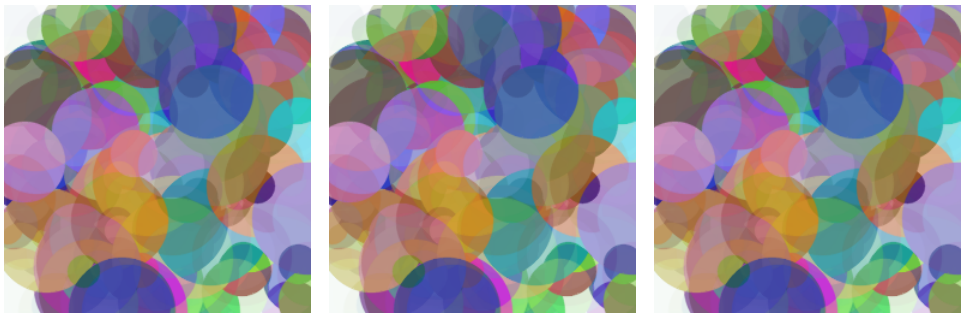
Introduciamo alcune definizioni che ci torneranno utili:

- 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 teniche e linguaggi di parallelizzazione. In Figura 1 è

mostrato un esempio del risultato ottenuto con $N = 10000$ e $n = 50$ usando i diversi metodi di parallelizzaione 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* generateCircles(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* generatePlanes(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 seuqnzialmente 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

    printf("%d %d %d %d\n", result.data[100], result.data[101], result.data[102], result.data[103]);
    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.
Poiché la generazione dei cerchi impiega tempi molto lunghi, abbiamo prima di tutto utilizzato OpenMP per velocizzare questo processo.

```
Circle* generateCirclesPar(std::size_t n, int width, int height, int minRadius, int maxRadius) {
    auto* circles = new Circle[n];
    std::mt19937 generator(std::random_device{}());

    std::uniform_int_distribution<int> colorDistribution(0, 255);
    std::uniform_int_distribution<int> pointXDistribution(1, width);
    std::uniform_int_distribution<int> pointYDistribution(1, height);
    std::uniform_int_distribution<int> radiusDistribution(minRadius, maxRadius);

#pragma omp parallel for default(none) shared(circles, generator)
    for (int i = 0; i < n; i++) {
        cv::Scalar color(colorDistribution(generator), colorDistribution(generator), colorDistribution(generator), 255);
        cv::Point center(pointXDistribution(generator), pointYDistribution(generator));
        int r = radiusDistribution(generator);
        circles[i] = Circle{color, center, r};
    }

    return circles;
}

cv::Mat* generatePlanesPar(std::size_t nPlanes, Circle circles[], std::size_t nCircles) {
    auto *planes = new cv::Mat[nPlanes];

#pragma omp parallel for default(none) shared(planes, circles) firstprivate(nPlanes, nCircles)
    for (int i = 0; i < nPlanes; i++) {
        planes[i] = TRANSPARENT_MAT;
        for (int j = 0; j < nCircles; j++) {
```

```
28              auto circle = circles[i * nCircles + j];
29              cv::circle(planes[i], circle.center, circle.r, circle.
    color, cv::FILLED, cv::LINE_AA);
30          }
31      }
32
33      return planes;
34  }
```

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. In Tabella 1 sono mostrati i tempi di esecuzione impiegati per la generazione variando il numero di piani e il numero di cerchi per piano.

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, poichè 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, ad 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 eseguibibile 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 ad 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(16, 16); // threads x block
3   dim3 grid(result.cols / block.x, result.rows/ block.y); // blocks
```

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

```
        auto idx = y * width + x;
        auto oneMinusAlpha = 1.0f - ALPHA;
        auto result = resultData[idx];

        for (int z = 0; z < nPlanes; z++) {
            auto idxP = z * width * height + idx;
            const auto &plane = planesData[idxP];

            result.x = result.x * oneMinusAlpha + plane.x * ALPHA;
            result.y = result.y * oneMinusAlpha + plane.y * ALPHA;
            result.z = result.z * oneMinusAlpha + plane.z * ALPHA;
            result.w = result.w * oneMinusAlpha + plane.w * ALPHA;
        }
        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

La macchina utilizzata per effettuare i test è dotata di:

- Processore Intel Core i5-8600K 3.60 GHz (6 core)
- 16 GB di RAM
- Scheda grafica NVIDIA GeForce GTX 1050 Ti 4 GB
- Sistema operativo Windows 11

6 Esperimenti e Risultati

I test sono stati condotti variando il numero di piani e le dimensioni 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.

Il numero di piani varia da 1000 a 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 da 100 a 1000, con un incremento di 100 piani.

Le versioni delle librerie utilizzate sono OpenCV 4.6.0 e CUDA 11.8.

7 Generazione

	$N = 100$		$N = 1000$		$N = 10000$	
	$n = 50$	$n = 500$	$n = 50$	$n = 500$	$n = 50$	$n = 500$
Sequential	TODO	TODO	TODO	TODO	TODO	TODO
Parallel	TODO	TODO	TODO	**TODO**	TODO	**TODO**

Tabella 1: Tempi impiegati per la generazione dei cerchi e dei piani

7.1 256x256

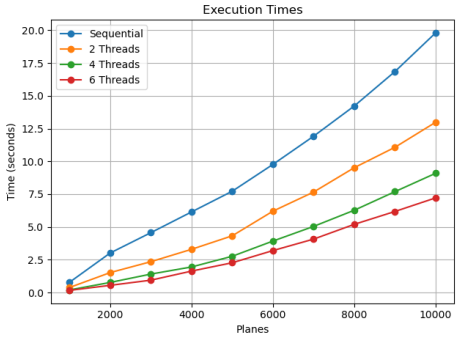


Figura 2: Tempi di OMP

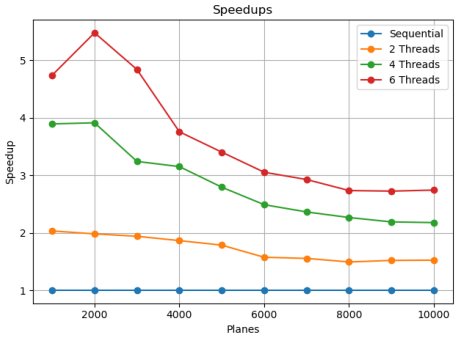


Figura 3: Speedup di OMP

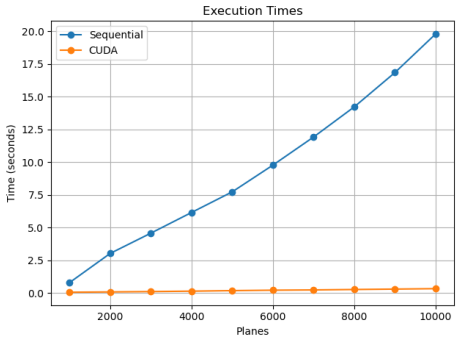


Figura 4: Tempi di CUDA

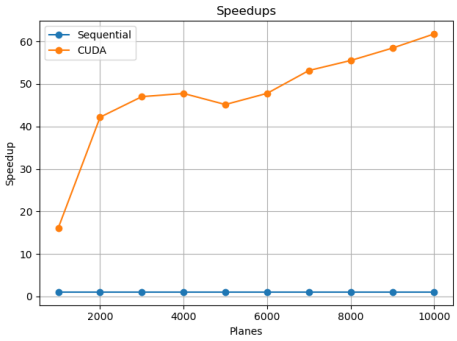


Figura 5: Speedup di CUDA

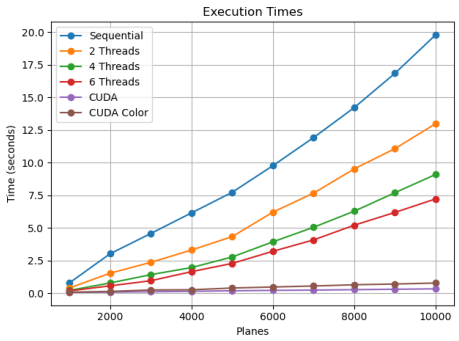


Figura 6: Confronto dei tempi

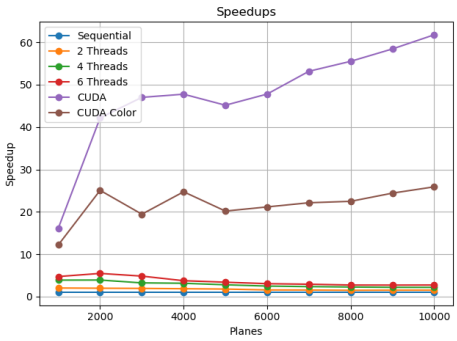


Figura 7: Confronto fra speedups

7.2 512x512

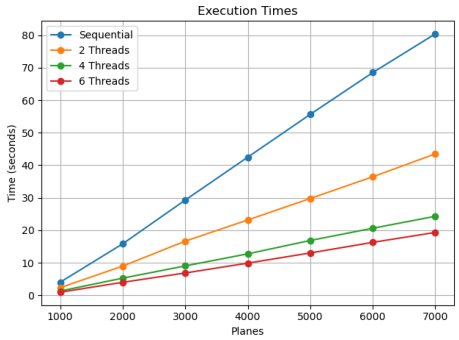


Figura 8: Tempi di OMP

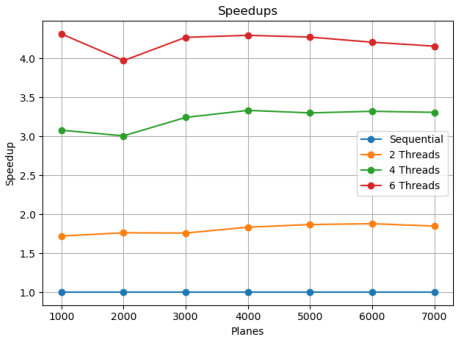


Figura 9: Speedup di OMP

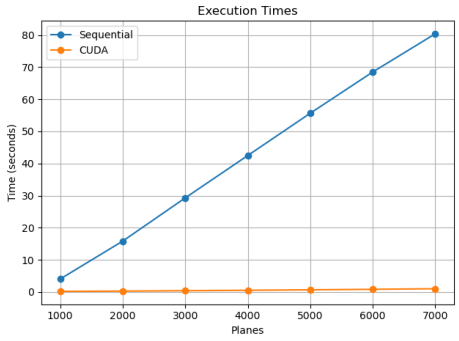


Figura 10: Tempi di CUDA

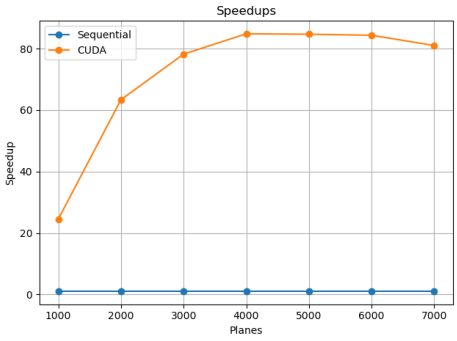


Figura 11: Speedup di CUDA

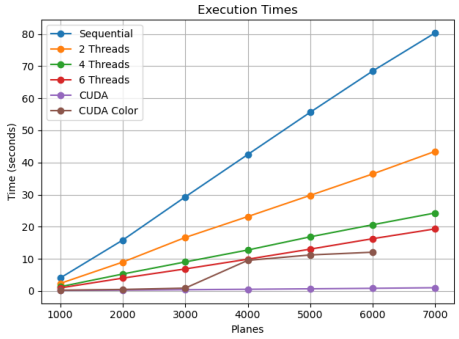


Figura 12: Confronto dei tempi

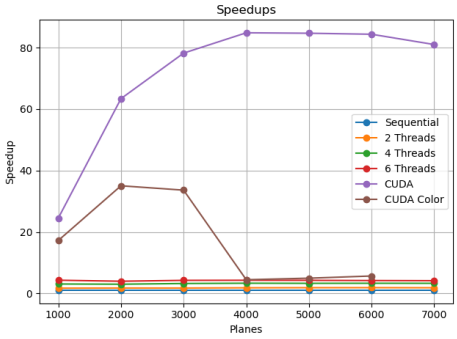


Figura 13: Confronto fra speedups

7.3 1024x1024

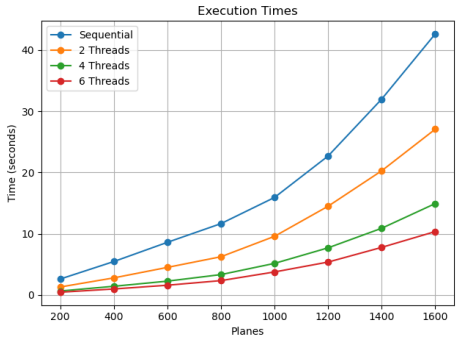


Figura 14: Tempi di OMP

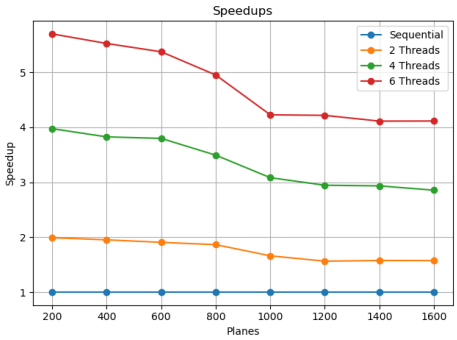


Figura 15: Speedup di OMP

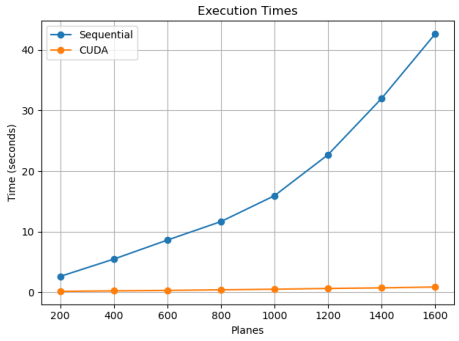


Figura 16: Tempi di CUDA

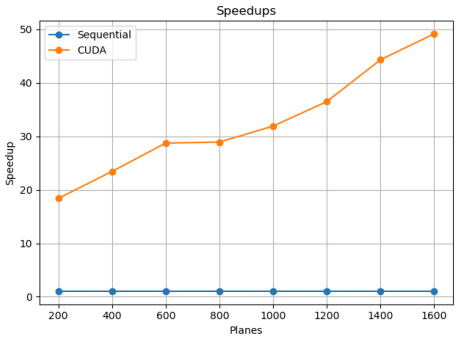


Figura 17: Speedup di CUDA

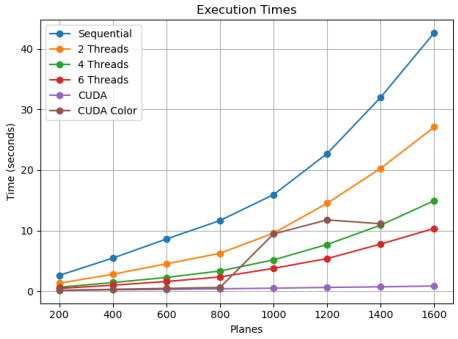


Figura 18: Confronto dei tempi

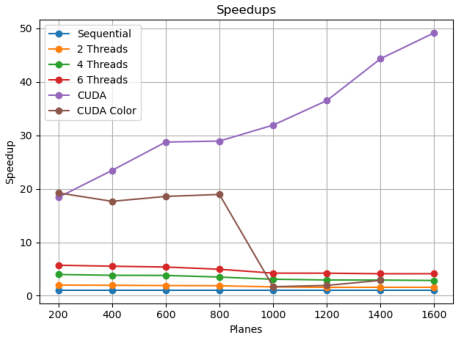


Figura 19: Confronto fra speedups

8 Conclusioni

In sintesi, come evidenziato dai test condotti, la parallelizzazione mediante CUDA si distingue per il notevole incremento di speedup riscontrato nelle diverse prove. Questo risultato è principalmente attribuibile alla progettazione della scheda grafica, ottimizzata per eseguire operazioni in parallelo e sfruttare appieno la sua potenza computazionale.

Inoltre, l'analisi dei test dimostra che l'incremento di velocità è più significativo all'aumentare del numero di piani. Ciò è dovuto alla capacità della scheda grafica di massimizzare la sua potenza di calcolo in relazione al numero crescente di piani.

È interessante notare che la versione parallela di OpenMP registra un aumento di velocità notevole rispetto alla versione sequenziale, soprattutto con l'incremento del numero di thread disponibili sulla macchina.