Renderer OpenMP
Problema di compositing

Parallel Programming for Machine Learning - Baiardi Lorenzo & Thomas Del Moro

Analisi del problema

- Ogni piano è composto da quattro canali di colore (RGBA, ovvero RGB con trasparenza), al fine di creare un effetto visivo di **trasparenza**.

- Su ciascun piano vengono disegnati *n* cerchi di colori diversi, dopodiché le immagini vengono sovrapposte dalla prima all'ultima.

- L'immagine finale, grazie alla trasparenza, sarà il risultato dell'insieme dei cerchi dell'ultimo piano, attraverso i quali saranno visibili quelli dei piani sottostanti.

Codice

Generazione sequenziale

Ciascun cerchio viene assegnato casualmente un valore RGB insieme alle sue coordinate spaziali (x, y) e al suo raggio. Successivamente, tutti i cerchi saranno memorizzati all'interno di un array chiamato "circles".

Su ciascun piano, inizialmente definito come una matrice trasparente, verrà disegnato un numero specifico di cerchi. Successivamente, tutti i piani saranno memorizzati all'interno di un array chiamato "planes".

```
Circle* sequentialGenerateCircles(std::size_t n, int width, int height, int minRadius, int maxRadius) {
    auto* circles = new Circle[n];
    std::mt19937 generator( Xx0: std::random_device{}());

    std::uniform_int_distribution<int> colorDistribution( Min0: 0,  Max0: 255);
    std::uniform_int_distribution<int> pointXDistribution( Min0: 1,  Max0: width);
    std::uniform_int_distribution<int> pointYDistribution( Min0: 1,  Max0: height);
    std::uniform_int_distribution<int> radiusDistribution( Min0: minRadius,  Max0: maxRadius);

    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* 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 : Circle  = circles[i * nCircles + j];
            cv::circle(planes[i], circle.center, circle.r, circle.color, cv::FILLED, cv::LINE_AA);
        }
    }
    return planes;
}
```

Codice

Generazione parallela

Per parallelizzare efficacemente la generazione dei cerchi e dei piani, sfruttiamo la clausola *#pragma omp parallel for*

Questa istruzione consente di distribuire in parallelo la creazione dei singoli cerchi e dei singoli piani all'interno dei rispettivi array

```
Circle* parallelGenerateCircles(std::size_t n, int width, int height, int minRadius, int maxRadius) {
    auto* circles = new Circle[n];
    std::mt19937 generator( Xx0: std::random_device{}());
    std::uniform_int_distribution<int> colorDistribution( Min0: 0,  Max0: 255);
    std::uniform_int_distribution<int> pointXDistribution( Min0: 1,  Max0: width);
    std::uniform_int_distribution<int> pointYDistribution( Min0: 1,  Max0: height);
    std::uniform_int_distribution<int> radiusDistribution( Min0: minRadius,  Max0: 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* parallelGeneratePlanes(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++) {
            auto circle : Circle  = circles[i * nCircles + j];
            cv::circle(planes[i], circle.center, circle.r, circle.color, cv::FILLED, cv::LINE_AA);
        }
    }
    return planes;
}
```

Risultati - Generazione Parallela

	$N = 100$		$N = 1000$		$N = 10000$	
	$n = 50$	$n = 500$	$n = 50$	$n = 500$	$n = 50$	$n = 500$
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

Codice

Renderer sequenziale

Per l'operazione di compositing dei piani è stata sviluppata una funzione facilmente parallelizzabile.

Per ogni pixel dell'immagine finale si esegue una somma pesata del valore di tale pixel su tutti i piani, separatamente per ognuno dei 4 canali.

```
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;
}
```

Codice

Renderer parallelo

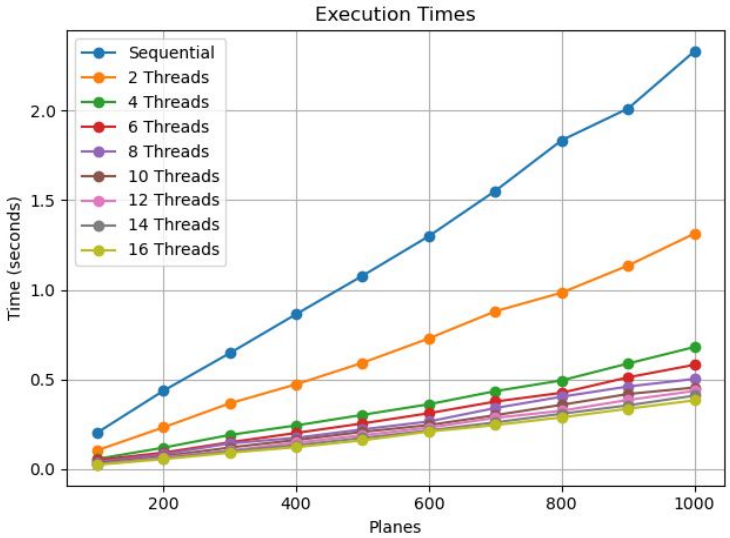
La funzione di compositing è stata parallelizzata con OpenMP

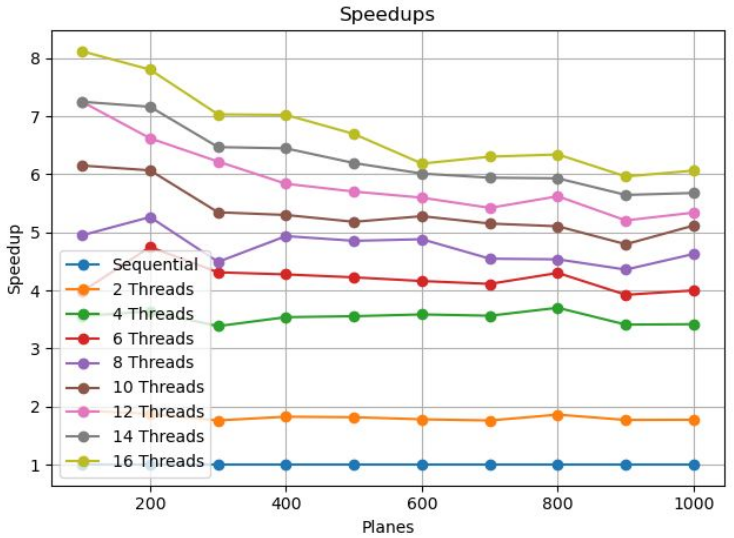
Con la direttiva *#pragma omp parallel for* è stato costruito un loop parallelo che può essere eseguito da più thread contemporaneamente

La clausola *collapse(2)* permette di parallelizzare sulle righe e sulle colonne della matrice

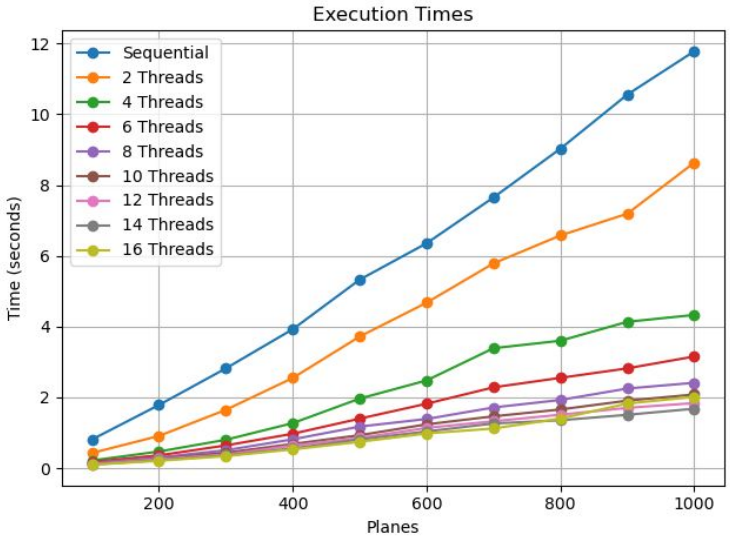
```
double parallelRenderer(cv::Mat planes[], std::size_t nPlanes) {
    cv::Mat result = TRANSPARENT_MAT;
    int cn = result.channels();

    // START
    double start = omp_get_wtime();

#pragma omp parallel for default(none) shared(result, planes) firstprivate(nPlanes, cn) collapse(2)
    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(PAR_IMG_PATH + std::to_string(nPlanes) + ".png", result);
    return time;
}
```

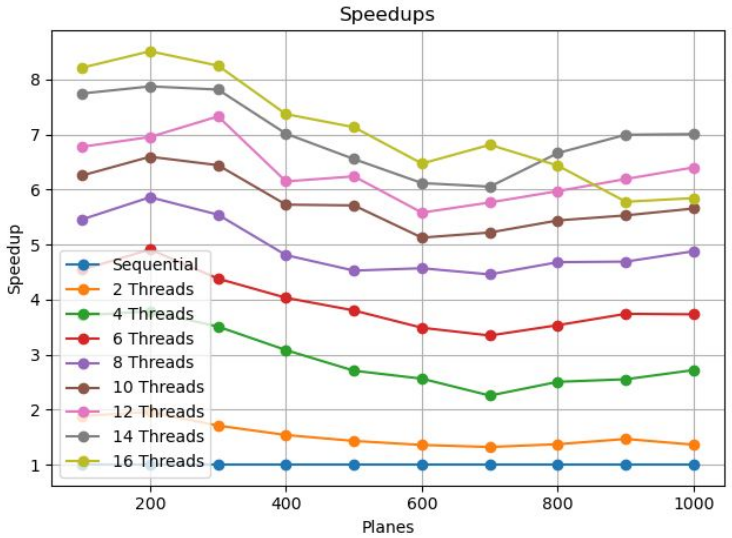
Risultati - Immagini 256x256



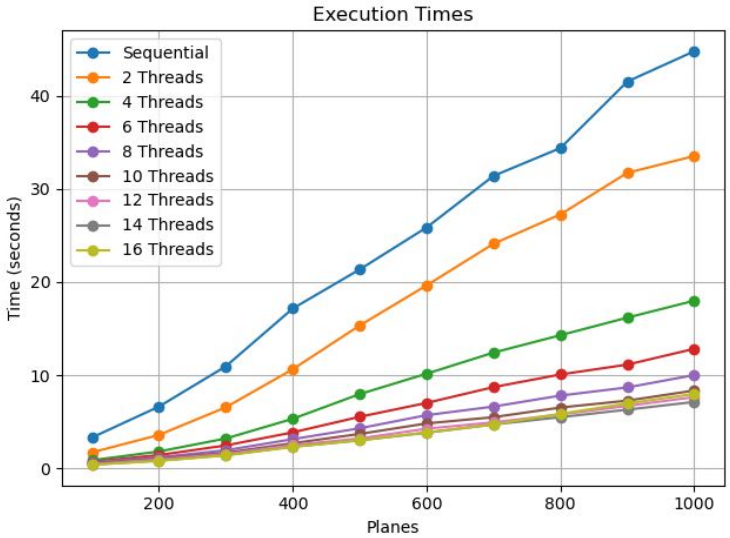


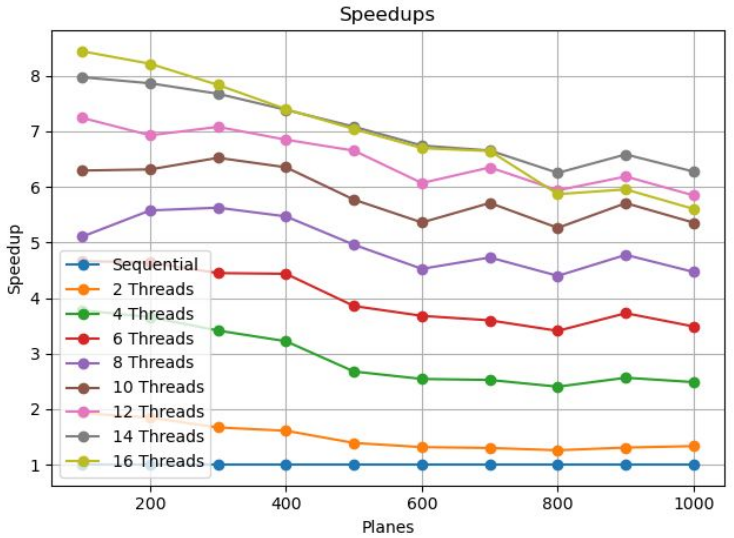
Risultati - Immagini 512x512





Risultati - Immagini 1024x1024





Conclusioni

- Come dimostrato dai test svolti, la parallelizzazione con OpenMP si rivela estremamente vantaggiosa per la gestione del problema di compositing dei piani, offrendo tempi di esecuzione molto più rapidi rispetto alla computazione sequenziale.
- Il numero di thread utilizzati è un fattore determinante con immagini di dimensioni contenute. I thread sembrano perdere leggermente efficacia quando si trovano a gestire immagini molto grandi.
- Al variare del numero di piani lo speedup rimane quasi costante, indipendentemente dalla grandezza delle immagini.