Problemi di Render : Parallel Programming for Machine Learning

Lorenzo Baiardi, Thomas Del Moro

10 12 2023

1 Introduzione

In questo documento, esamineremo attentamente l'efficacia dell'applicazione di diversi metodi di parallelizzazione su problematiche comuni, analizzando in dettaglio i tempi di esecuzione associati. Concentreremo la nostra valutazione sul metodo del compositing tra piani, utilizzando la libreria grafica OpenCV come strumento principale per condurre le nostre analisi.

2 Analisi del problema

Ciascun piano è caratterizzato da quattro canali di colore (RGBA), e su ogni piano saranno disegnati casualmente n cerchi. Nel processo di compositing dei piani, sarà implementato un effetto di trasparenza, il quale dipenderà dalla posizione relativa di ciascun piano all'interno della sommatoria. Questa strategia mira a creare un effetto visivo di profondità. L'immagine risultante sarà salvata nel formato PNG e si focalizzerà principalmente sul piano posizionato più in alto, evidenziando così principalmente i cerchi associati a esso.

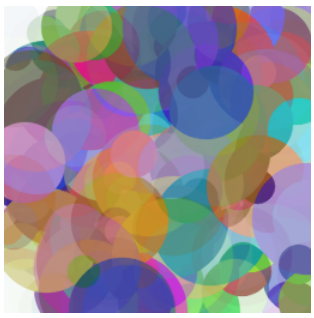

Figura 1: Sequenziale

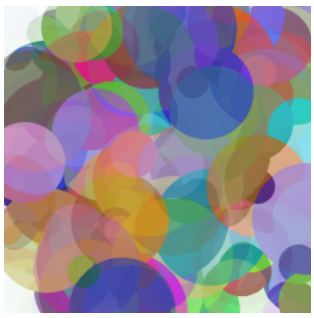

Figura 2: OMP

Figura 3: CUDA

Figura 4: Immagini di esempio

3 Metodi di parallelizzazione

Per parallelizzare la problematica, abbiamo optato per l'adozione di due approcci distinti. Nel primo scenario, abbiamo impiegato la libreria OPENMP, esplorando le variazioni nel numero di thread. Nel secondo caso, abbiamo invece sfruttato il linguaggio di programmazione CUDA per eseguire calcoli in parallelo sulla scheda grafica.

```
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 *src2 = &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) +
                            src2->data[i * src2->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;
}
```

3.1 OPENMP

L'approccio fondamentale consiste nell'assegnare a ciascun thread la responsabilità di gestire la sommatoria di un singolo pixel per ogni matrice. Questo permette di mantenere l'ordine dei piani, mentre allo stesso tempo incrementa la velocità di rendering complessiva. Ogni thread, di conseguenza, elaborerà il pixel risultante e procederà successivamente al calcolo del pixel successivo una volta completata l'operazione corrente.

```
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 *src2 = &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) +
                            src2->data[i * src2->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;
}
```

3.2 CUDA

Attraverso un attento calcolo dei parametri di grid e block, miriamo a ottimizzare l'allocazione delle risorse sulla scheda grafica. Definendo una suddivisione efficiente del lavoro tra i thread e i blocchi, intendiamo massimizzare la parallelizzazione, sfruttando appieno la capacità computazionale offerta dalla scheda grafica. Questo approccio mirato a una gestione ottimale delle risorse è essenziale per garantire prestazioni elevate nel contesto del calcolo parallelo mediante CUDA.

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

    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;
```

```
12              const auto &plane = planesData[idxP];
13
14              result.x = result.x * oneMinusAlpha + plane.x *
    ALPHA;
15              result.y = result.y * oneMinusAlpha + plane.y *
    ALPHA;
16              result.z = result.z * oneMinusAlpha + plane.z *
    ALPHA;
17              result.w = result.w * oneMinusAlpha + plane.w *
    ALPHA;
18          }
19          resultData[idx] = result;
20      }
21  }
```

4 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

5 Tests

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.

5.1 256x256

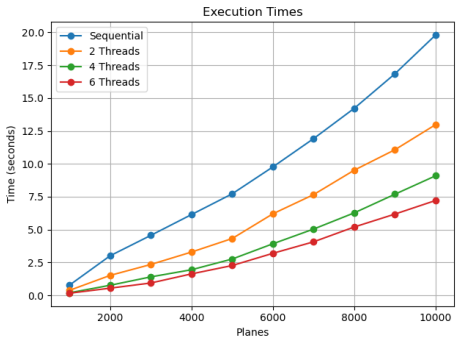

Figura 5: Tempi di OMP

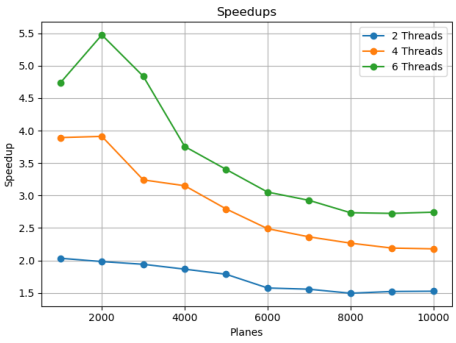

Figura 6: Speedup di OMP

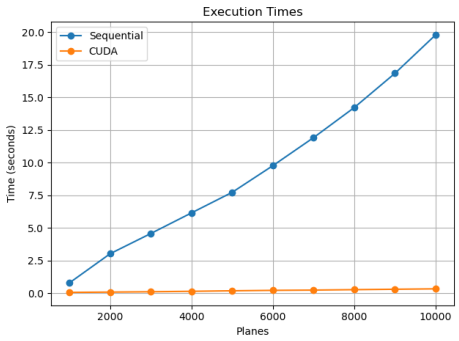

Figura 7: Tempi di CUDA

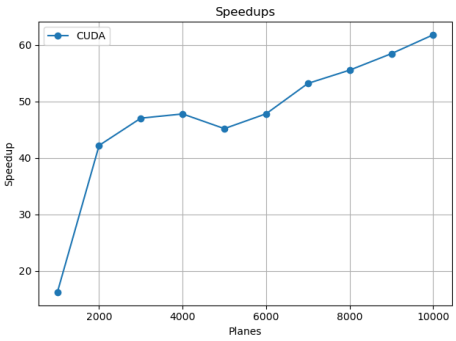

Figura 8: Speedup di CUDA

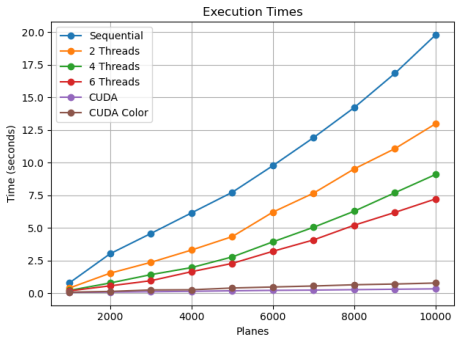

Figura 9: Confronto dei tempi

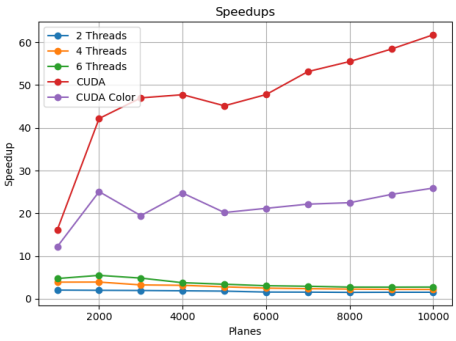

Figura 10: Confronto fra speedups

5.2 512x512

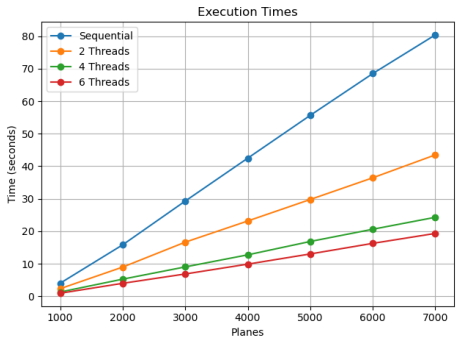

Figura 11: Tempi di OMP

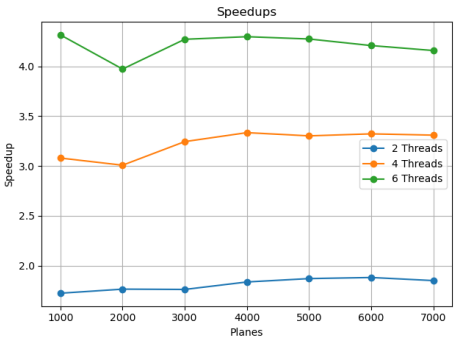

Figura 12: Speedup di OMP

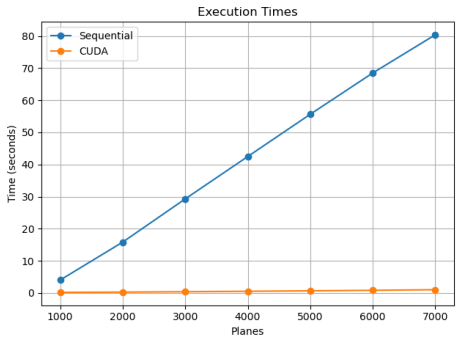

Figura 13: Tempi di CUDA

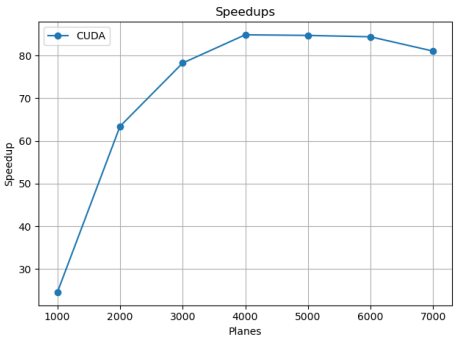

Figura 14: Speedup di CUDA

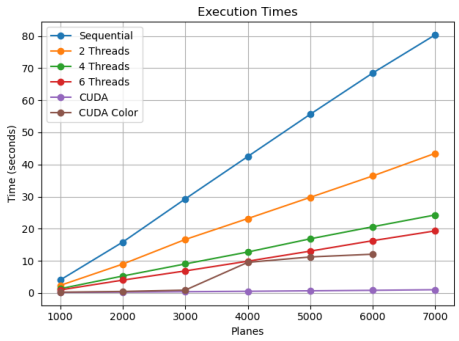

Figura 15: Confronto dei tempi

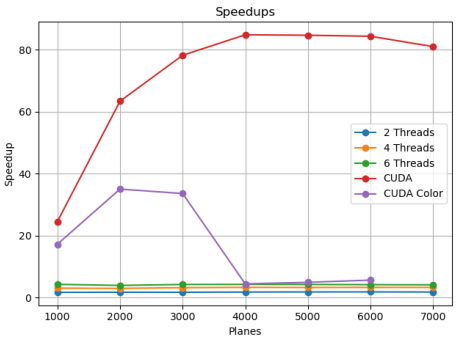

Figura 16: Confronto fra speedups

5.3 1024x1024

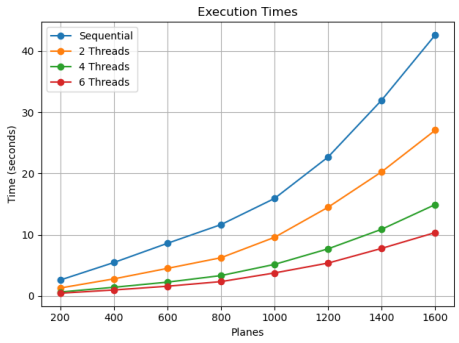

Figura 17: Tempi di OMP

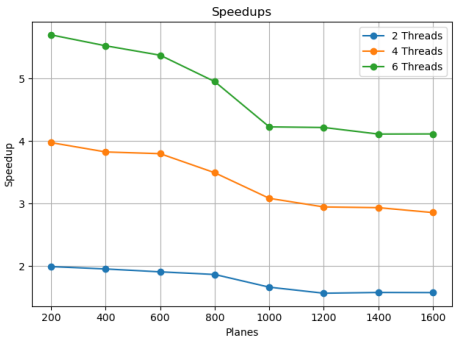

Figura 18: Speedup di OMP

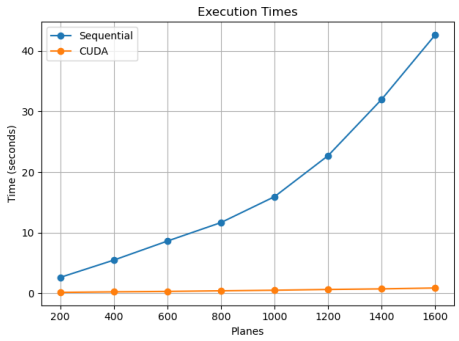

Figura 19: Tempi di CUDA

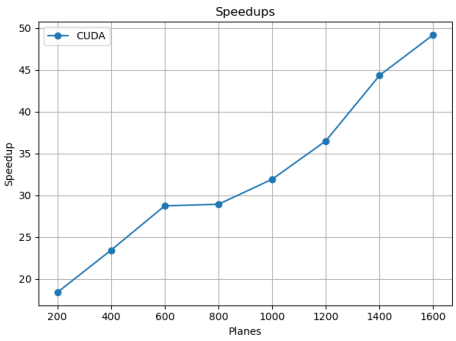

Figura 20: Speedup di CUDA

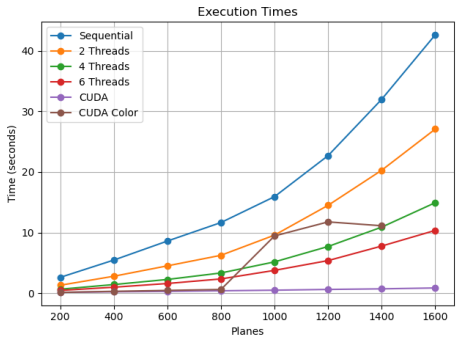

Figura 21: Confronto dei tempi

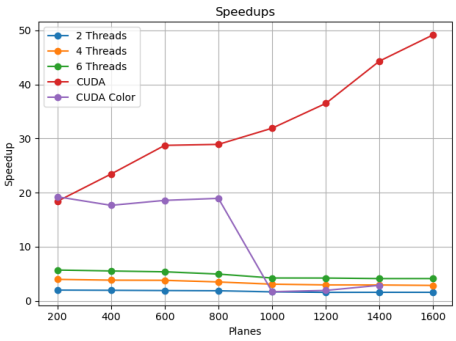

Figura 22: Confronto fra speedups

6 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.