Renderer Cuda
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

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 Cuda

Costruiamo una *grid* le cui dimensioni sono state ottimizzate sulla base di risultati empirici e dipendono dalle dimensioni dell'immagine considerata.

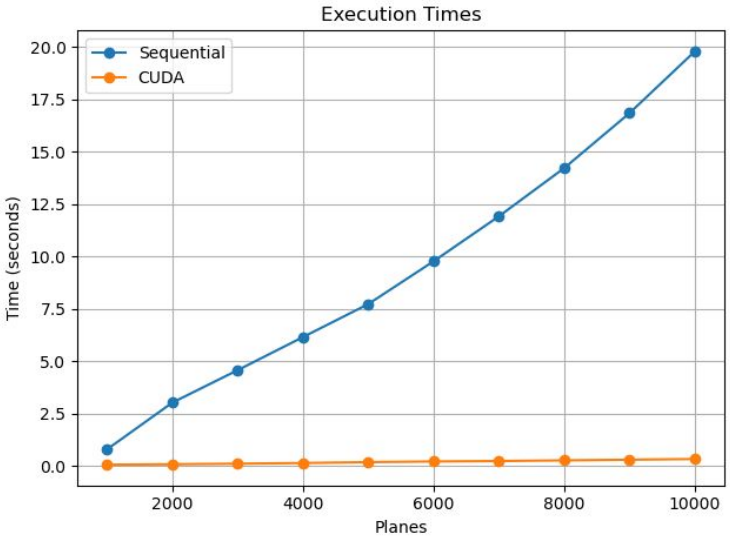
Ogni thread esegue la somma su tutti i piani del pixel dell'immagine finale ad esso associato, per tutti e 4 i canali.

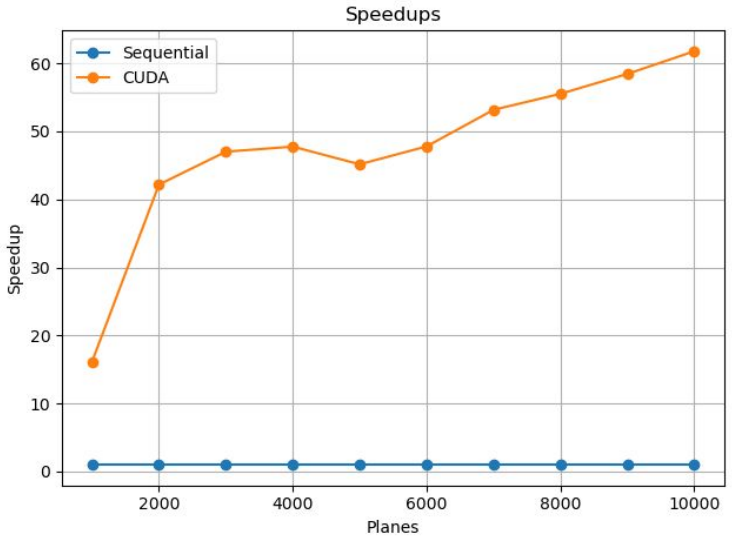
```
// GRID AND BLOCK DIMENSIONS
dim3 block(blockSize, blockSize); // threads x block
dim3 grid(result.cols / block.x, result.rows/ block.y); // blocks

// CUDA KERNEL
cudaKernelCombinePlanes<<<grid, block>>>(d_resultData, d_planesData,
                                         result.cols, result.rows, (int) nPlanes);

__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 :double = 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;
    }
}
```

Risultati - Dimensione dei thread-blocks

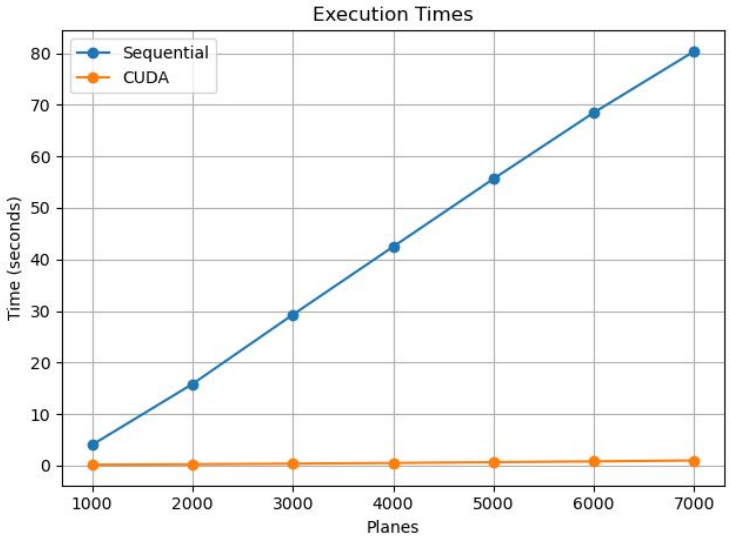
	Dimensione dell'immagine					
	$256x256$		$512x512$		$1024x1024$	
	$N = 500$	$N = 5000$	$N = 500$	$N = 5000$	$N = 500$	$N = 2000$
$8x8$	0.024	0.189	0.086	0.884	0.252	2.507
$16x16$	**0.021**	**0.157**	0.073	**0.645**	**0.249**	1.214
$32x32$	0.022	0.161	**0.066**	0.657	**0.249**	**1.107**

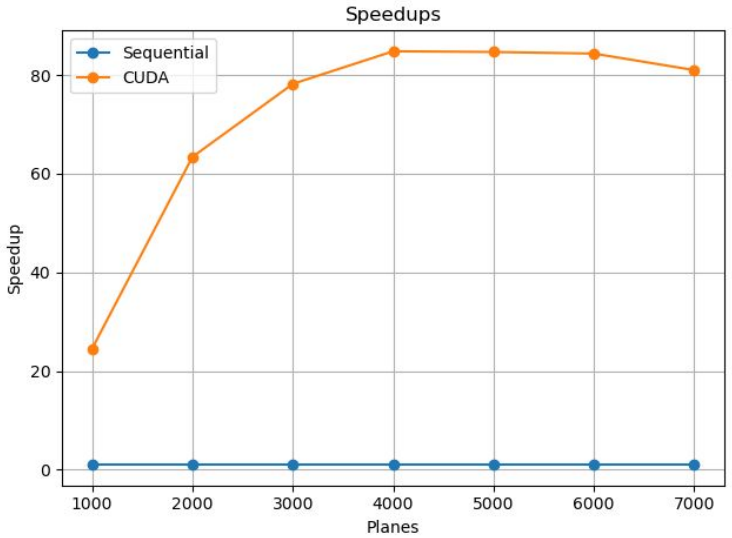
Risultati - Immagini 256x256



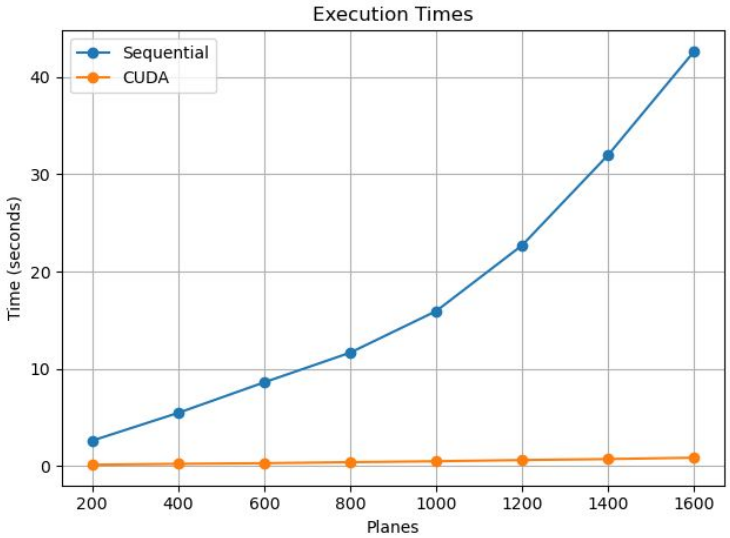


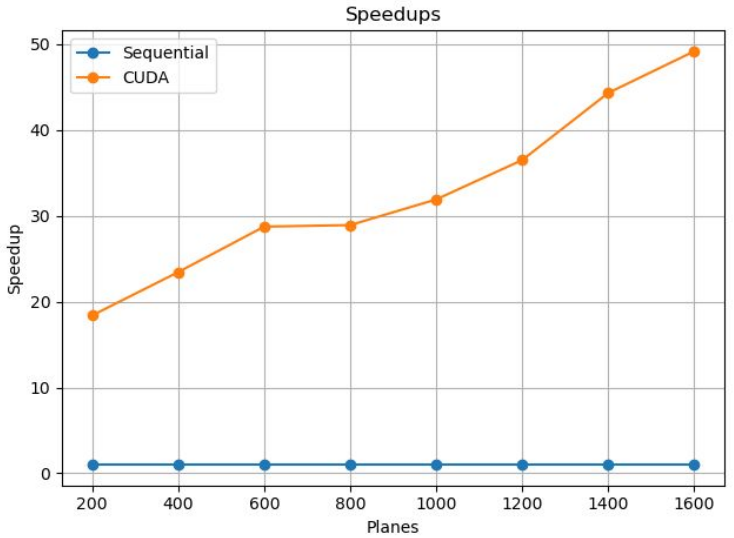
Risultati - Immagini 512x512





Risultati - Immagini 1024x1024

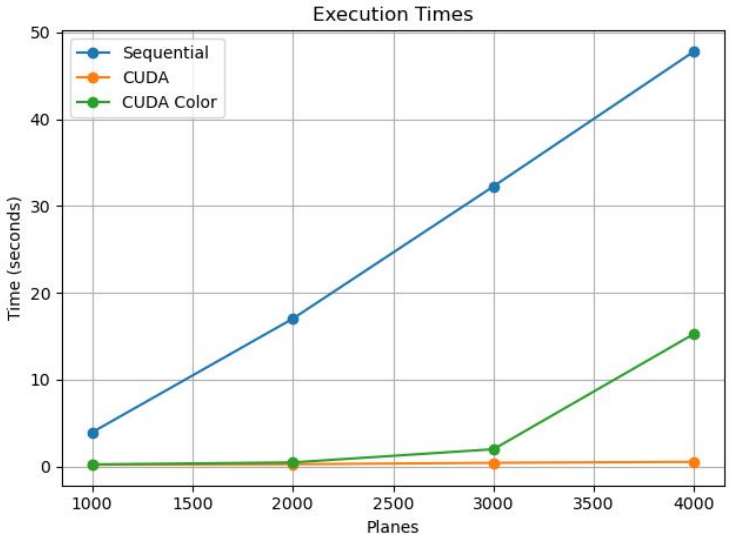


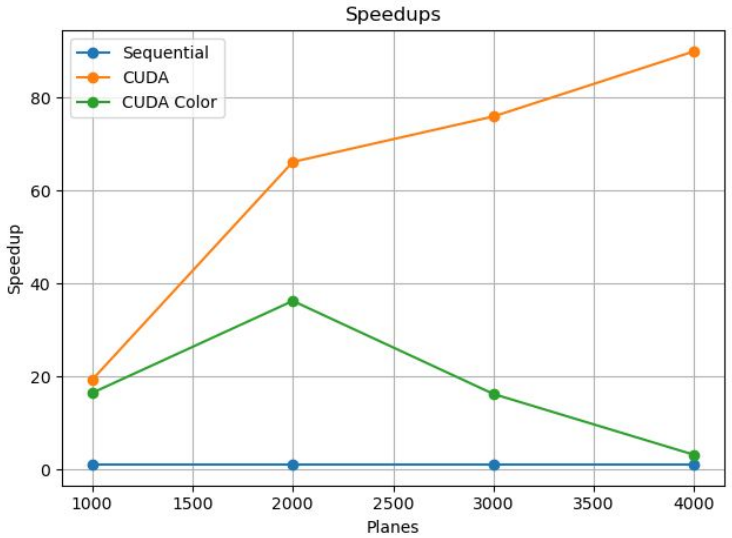


Codice Cuda (altra versione)

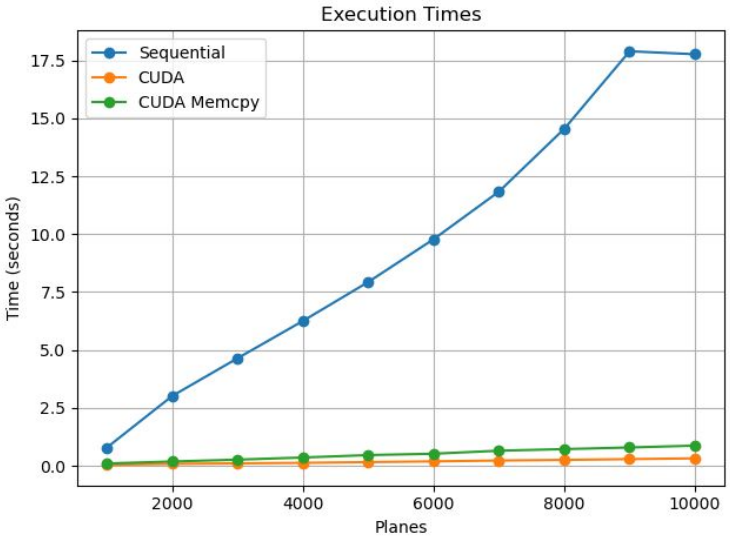
```
__global__ void cudaKernelCombinePlanesColor(uchar4* d_resultData, const uchar* d_planesData,
                                             const int width, const int height, const int nPlanes) {

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

    if (x < width && y < height) {
        auto oneMinusAlpha :double = 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]};
    }
}
```

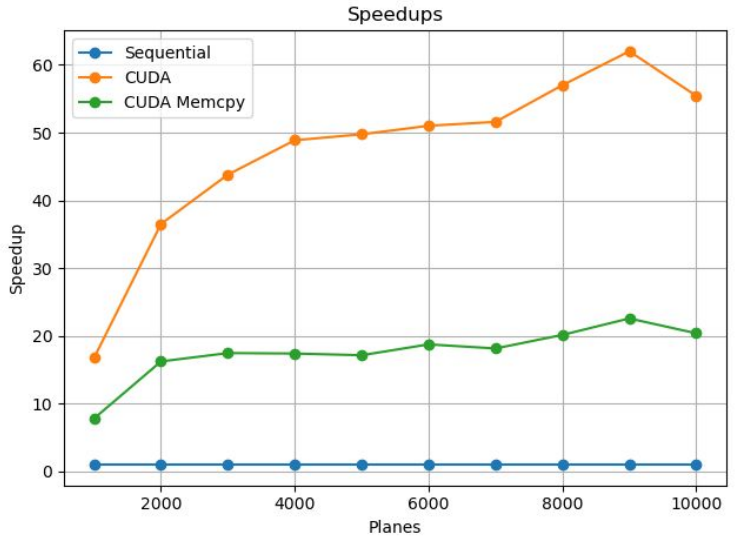
Risultati - Immagini 512x512 Cuda Color





Risultati - Immagini 256x256 Cuda Memcpy





Conclusioni

Come dimostrato dai test svolti, la parallelizzazione si rivela estremamente vantaggiosa per la gestione del problema di compositing dei piani, offrendo tempi di esecuzione molto più rapidi rispetto alla computazione sequenziale.

In particolare, l'approccio CUDA si distingue per la sua efficacia, consentendo uno speedup molto significativo, soprattutto su immagini 512x512.

Grazie all'ottimizzazione della dimensione dei thread-blocks abbiamo migliorato ulteriormente le performance della versione parallelizzata.

In tutti i casi, sia all'aumentare della dimensione dell'immagine che del numero di piani, si ha un progressivo vantaggio nell'utilizzo della GPU rispetto all'approccio sequenziale.