Problemi di Render : Parallel Programming for Machine Learning

Lorenzo Baiardi, Thomas Del Moro

10 12 2023

1 Introduzione

In questo elaborato verificheremo l'efficacia dell'utilizzo di vari metodi di parallelizzazione in problemi comuni, studiandone i tempi di esecuzione. In particolare valutiamo il metodo del compositing tra piani tramite l'utilizzo della libreria grafica OpenCV.

2 Analisi del problema

Ogni piano ha 4 canali di colore (RGBA) e per ogni piano verranno disegnati n cerchi in maniera del tutto casuale. Durante la fase di compositing dei piani, verrà applicato un effetto di trasparenza in base alla posizione del piano all'interno della sommatoria, in modo da ottenere un effetto di profondità. L'immagine risultante sarà salvata come file PNG e metterà in risalto principalmente il piano, quindi cerchi, più in alto.

Figura 1: Sequenziale

Figura 2: OMP

Figura 3: CUDA

Figura 4: Immagini di esempio

3 Metodi di parallelizzazione

Per poter parallelizzare il problema abbiamo deciso di utilizzare due approcci differenti, nel primo caso abbiamo utilizzato la libreria OPENMP, al variare del numero di thread, e nel secondo caso abbiamo utilizzato il linguaggio di programmazione CUDA per il calcolo parallelo su scheda grafica.

```
1 double sequentialRenderer(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       for (int i = 0; i < result.rows; i++) {
9           for (int j = 0; j < result.cols; j++) {
10              for (int z = 0; z < nPlanes; z++) {
11                  cv::Mat *src2 = &planes[z];
12                  for (int c = 0; c < cn; c++)
13                      result.data[i * result.step + cn * j + c] =
14                              result.data[i * result.step + j *
    cn + c] * (1 - ALPHA) +
15                              src2->data[i * src2->step + j * cn
    + c] * (ALPHA);
16              }
17          }
18      }
19
20      double time = omp_get_wtime() - start;
21      // END
22
23      cv::imwrite(SEQ_IMG_PATH + std::to_string(nPlanes) + ".png"
    , result);
24      return time;
25  }
```

3.1 OPENMP

L'idea di fondo è quella che ogni thread gestisca la sommatoria di un singolo pixel per ogni matrice, in modo da preservare l'ordinamento dei piani ma aumentandone la velocità di render. Di conseguenza ogni thread calcolerà il pixel risultante e successivamente, al termine di esso, passerà al successivo pixel.

```
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++) {
```

```
                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

In questo caso la parte di parallelizzazione si svilupperà principalmente nel calcolo di grid e block da utilizzare, in modo da poter sfruttare al meglio la potenza di calcolo della scheda grafica.

```
__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 resultX = resultData[idx].x;
        auto resultY = resultData[idx].y;
        auto resultZ = resultData[idx].z;
        auto resultW = resultData[idx].w;

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

            resultX = resultX * oneMinusAlpha + plane.x * ALPHA
    ;
            resultY = resultY * oneMinusAlpha + plane.y * ALPHA
    ;
```

```
            resultZ = resultZ * oneMinusAlpha + plane.z * ALPHA;
            resultW = resultW * oneMinusAlpha + plane.w * ALPHA;
        }

        resultData[idx] = {resultX, resultY, resultZ, resultW};
    }
}
```

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 quindi effettuati al variare del numero dei piani e dalle dimensioni delle immagini. Tutti i piani e i vari cerchi vengono generati precedentemente all'esecuzione del test, in modo da poter avere un confronto più accurato tra i vari metodi di parallelizzazione. Le dimensioni delle immagini utilizzate sono 256x256, 512x512 e 1024x1024. Il numero di piani partono da 1000 e arrivano a 10000, con un incremento di 1000 piani per ogni test. La versione di OpenCV utilizzata è la 4.6.0, mentre la versione di CUDA è la 11.8.

5.1 256x256

5.2 512x512

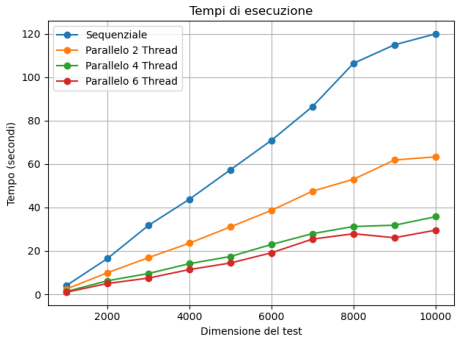

Figura 5: Tempi di OMP

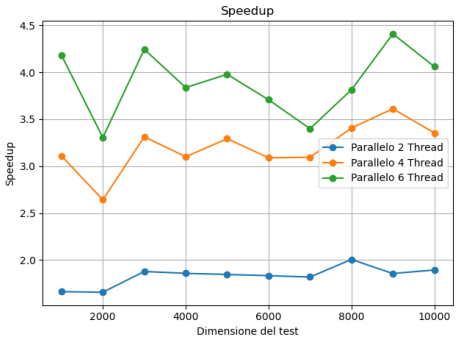

Figura 6: Speedup di OMP

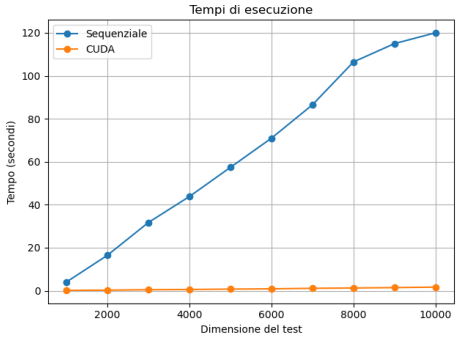

Figura 7: Tempi di CUDA

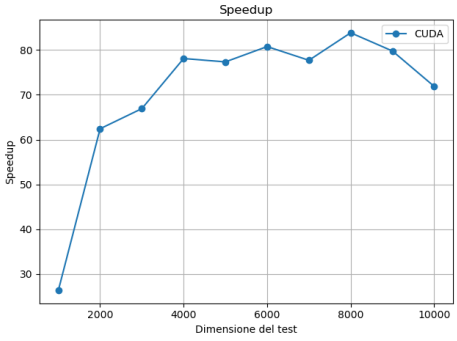

Figura 8: Speedup di CUDA

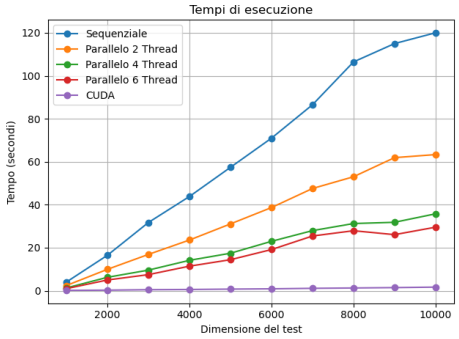

Figura 9: Confronto dei tempi

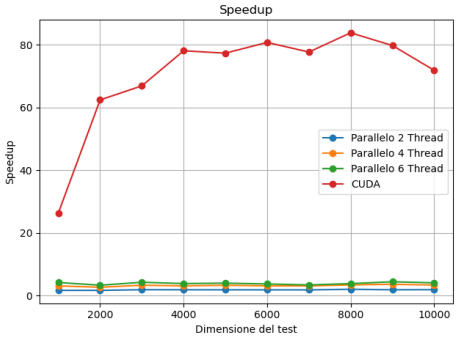

Figura 10: Confronto fra speedups

5.3 1024x1024

6 Conclusioni

In conclusione, come è possibile vedere dai vari test effettuati e come ci si potesse aspettare, l'operazione di parallelizzazione CUDA risulti quello che ottiene un maggior incremento a livello di speedUp nei vari test effettuati. Questo è dovuto principalmente al fatto che la scheda grafica è stata progettata per effettuare operazioni parallele, quindi è possibile sfruttare al meglio la potenza di calcolo della scheda grafica. Inoltre, come è possibile vedere dai vari test effettuati, l'incremento di velocità è maggiore al crescere del numero di piani, questo è dovuto al fatto che la scheda grafica riesce a sfruttare al meglio la sua potenza di calcolo al crescere del numero di piani. E' da notare anche come la versione parallela di OpenMP riesca a ottenere un incremento di velocità maggiore rispetto alla versione sequenziale, specialmente all'aumentare del numero di thread che la macchina dispone.

7 Risultati

Test	TSeq	TPar	SpeedUp
1000	4.031	2.423	1.664
2000	16.495	9.949	1.658
3000	31.754	16.896	1.879
4000	43.843	23.582	1.859
5000	57.414	31.088	1.847
6000	70.984	38.698	1.834
7000	86.536	47.558	1.820
8000	106.463	53.034	2.007
9000	114.997	61.922	1.857
10000	120.016	63.340	1.895

(a) Risultati parallelo 2

Test	TSeq	TPar	SpeedUp
1000	4.031	1.297	3.109
2000	16.495	6.244	2.642
3000	31.754	9.585	3.313
4000	43.843	14.141	3.100
5000	57.414	17.439	3.292
6000	70.984	22.983	3.089
7000	86.536	27.962	3.095
8000	106.463	31.255	3.406
9000	114.997	31.843	3.611
10000	120.016	35.780	3.354

(b) Risultati parallelo 4

Test	TSeq	TPar	SpeedUp
1000	4.031	0.964	4.182
2000	16.495	4.991	3.305
3000	31.754	7.483	4.244
4000	43.843	11.426	3.837
5000	57.414	14.432	3.978
6000	70.984	19.149	3.707
7000	86.536	25.457	3.399
8000	106.463	27.917	3.814
9000	114.997	26.086	4.408
10000	120.016	29.564	4.060

(c) Risultati parallelo 6

Test	TSeq	TCuda	SpeedUp
1000	4.031	0.153	26.329
2000	16.495	0.264	62.419
3000	31.754	0.475	66.915
4000	43.843	0.561	78.102
5000	57.414	0.742	77.349
6000	70.984	0.879	80.771
7000	86.536	1.114	77.706
8000	106.463	1.270	83.846
9000	114.997	1.441	79.783
10000	120.016	1.670	71.877

(d) Risultati CUDA

Tabella 1: Confronto tra i risultati di parallelismo