UNIVERSITÀ
DEGLI STUDI
FIRENZE

Parallel Mean Shift Clustering

Nicolò Pollini, Francesco Fantechi

22/06/2023

UNIVERSITÀ DEGLI STUDI FIRENZE

Indice

UNIVERSITÀ DEGLI STUDI FIRENZE

Obiettivo

- Algoritmo MeanShift sequenziale
- Parallelizzazione con OpenMP
- Parallelizzazione con CUDA
- Analisi e confronto dei risultati

UNIVERSITÀ DEGLI STUDI FIRENZE

Implementazione

- Segmentazione di immagini PPM a colori
  - Spazio di clustering 5-dimensionale
- Linguaggio C++ per OpenMP e per CUDA
- Codice versionato su GitHub:
  - `https://github.com/francesco-ftk/Parallel-Mean-Shift-OpenMP`
  - `https://github.com/francesco-ftk/Parallel-Mean-Shift-CUDA`
- Test eseguiti su 3 macchine e 2 OS differenti

UNIVERSITÀ DEGLI STUDI FIRENZE

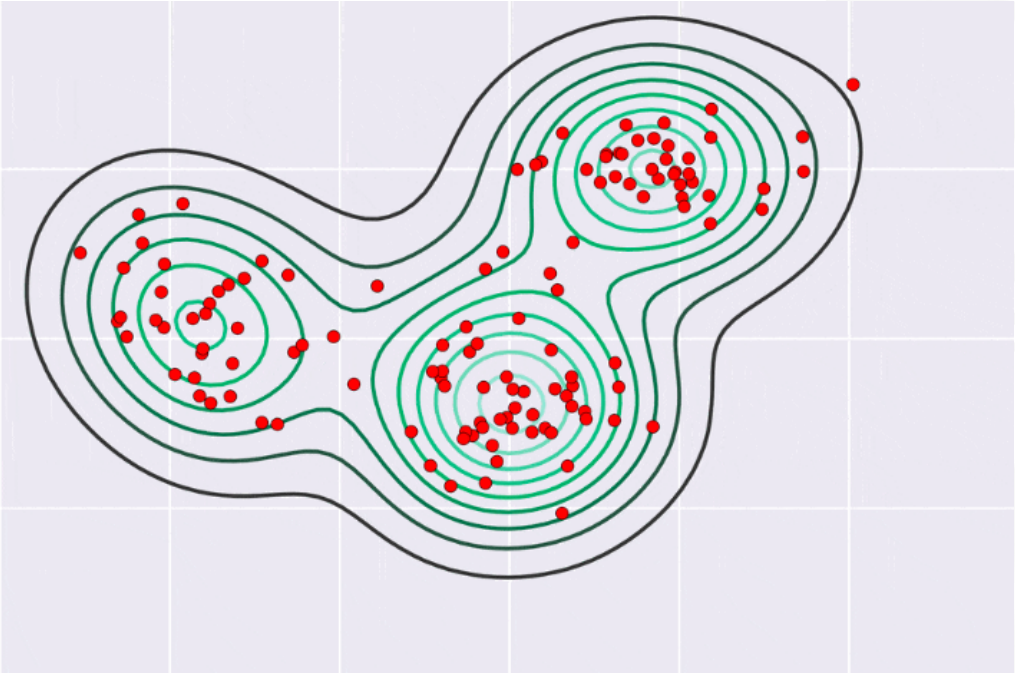
Implementazione

Machine 1				
OS	CPU	Number of Core with Hyper-Threading	RAM	GPU
Windows 10	Intel(R) Core(TM) i7-8750H	12	16 GB	NVIDIA GeForce GTX 1050 Ti
Machine 2				
OS	CPU	Number of Core with Hyper-Threading	RAM	GPU
Ubuntu 20.04.5	Intel(R) Core(TM) i7-1165G7	8	16 GB	NVIDIA GeForce MX350
Machine 3				
OS	CPU	Number of Core with Hyper-Threading	RAM	GPU
Windows 10	Intel(R) Core(TM) i7-7700K	8	32 GB	NVIDIA GeForce GTX 1080 Ti

Algoritmo MeanShift

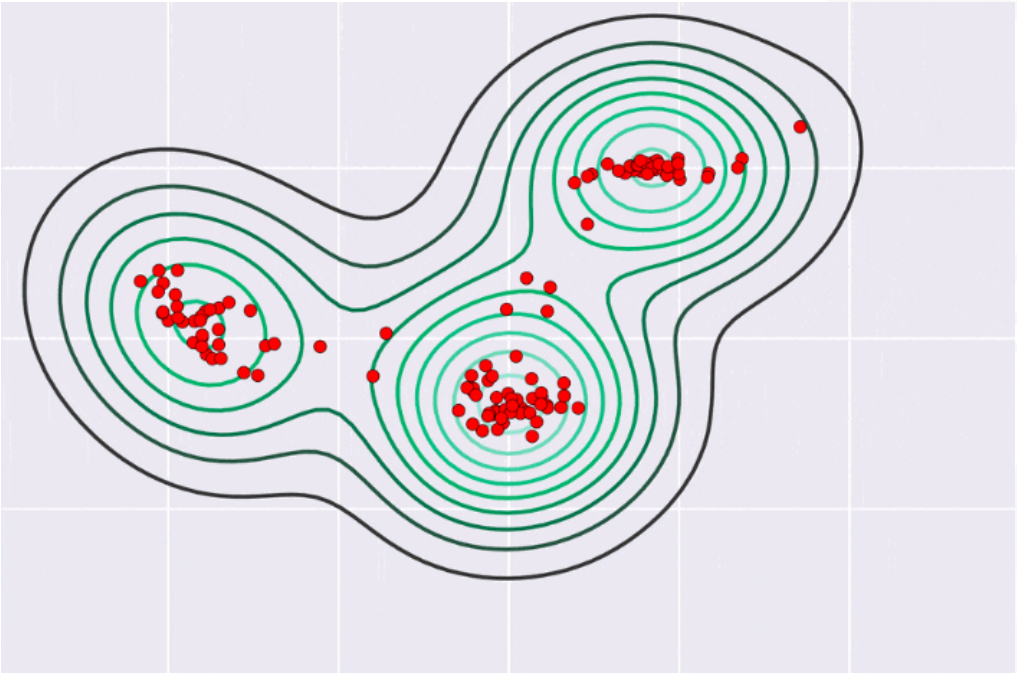
- Clustering N-dimensionale
- Applicazione del metodo di discesa del gradiente
- Differenze con K-Means:
  - no inizializzazione casuale,
  - rilevamento del numero di cluster,
  - parametri bandwidth e ϵ,
  - complessità $O(n^2)$

Algoritmo MeanShift



- Inizializzazione dei centroidi
- Filtraggio dei punti all'interno del supporto
- Calcolo dello scostamento medio

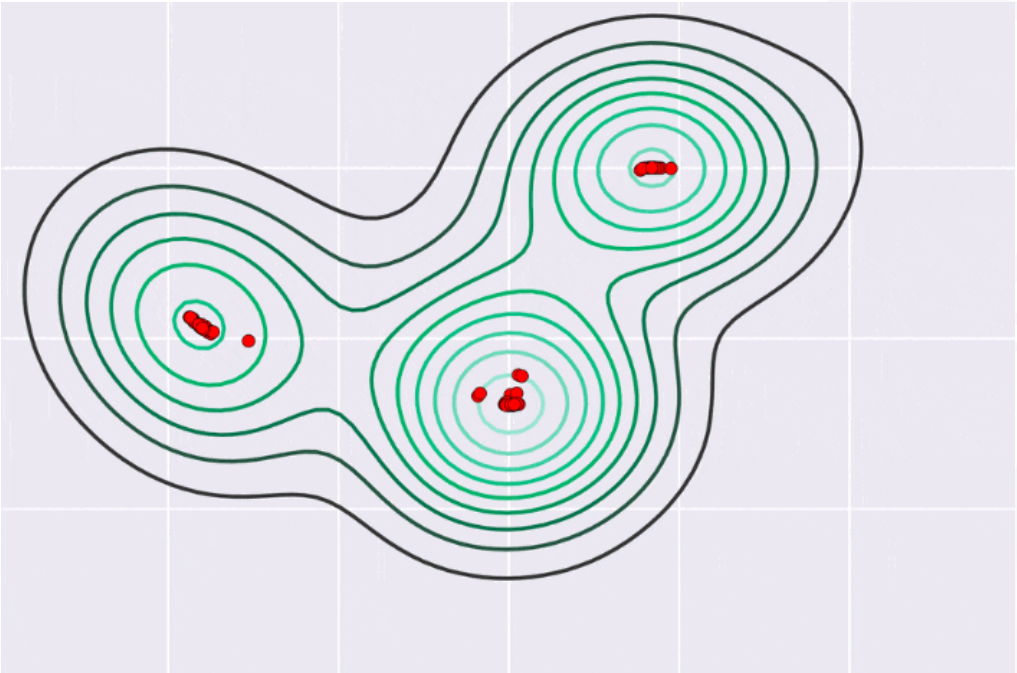
Algoritmo MeanShift



- Aggiornamento dei centroidi
- Confronto scostamento con ϵ

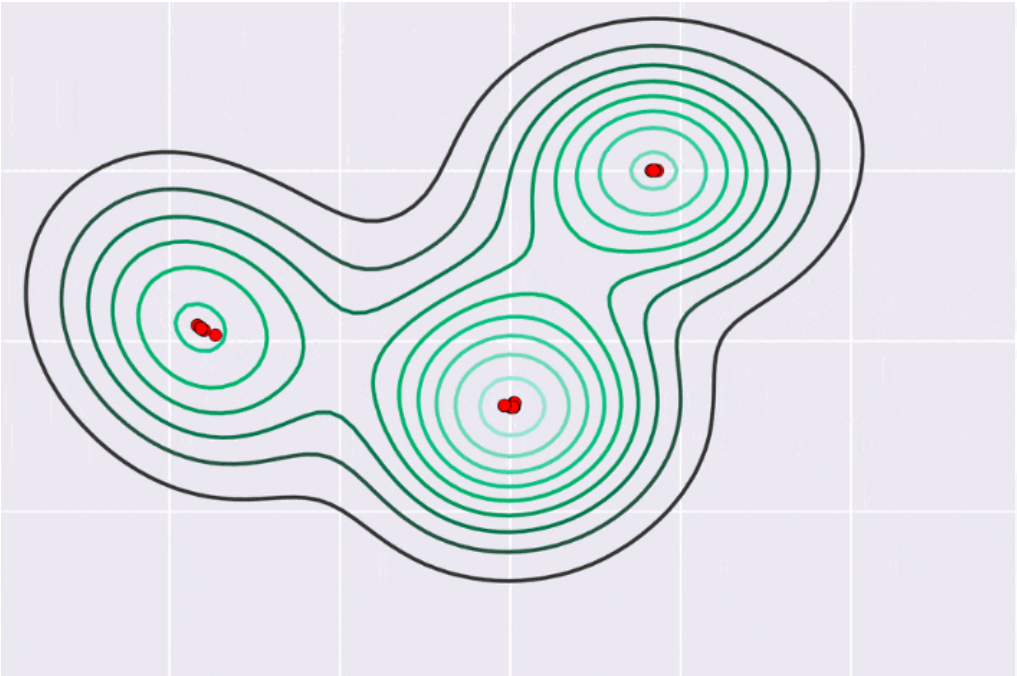
UNIVERSITÀ DEGLI STUDI FIRENZE

Algoritmo MeanShift



- Iterazione se scostamento $> \epsilon$

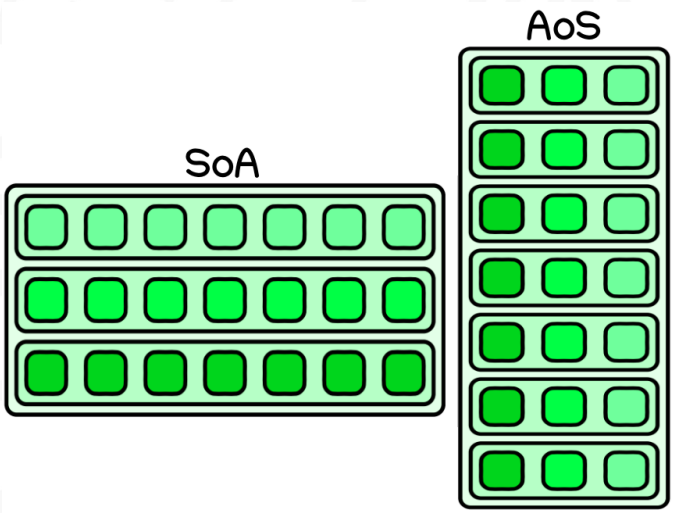
Algoritmo MeanShift



- Termine iterazioni
- Associazione delle mode
- Raggruppamento in cluster

Versione sequenziale

- AoS:
  - punti collocati sequenzialmente in un unico vettore,
  - punti rappresentati da una Struct di 5 campi
  - dati localmente vicini, accesso più semplice.
- SoA:
  - punti distribuiti su 5 array,
  - struttura più complessa, ma meglio allineabile alla cache,
  - accesso coalesced, vantaggi su versioni parallelizzate



Parallelizzazione

- Calcolo delle mode:
  - imbarazzantemente parallela,
  - complessità $O(n^2)$
    $\rightarrow$ parallelizzato
- Raggruppamento in cluster:
  - richiede collaborazione,
  - complessità O(n)
    $\rightarrow$ non parallelizzato

OpenMP

- Framework per parallelizzazione in modalità implicit threading
- Creazione di thread e parallelizzazione con direttive pragma
- Divisione del carico di lavoro con paradigma fork-join

```
// compute the means
#pragma omp parallel default(none) shared(points, means, modes) firstprivate(epsilon,
{
    #pragma omp for
    for (int i = 0; i < nOfPoints; ++i)
    {
        // initialize the mean on the current point
        float mean[CHANNELS];
        for (int k = 0; k < CHANNELS; ++k) { mean[k] = points[i * CHANNELS + k]; }
```

UNIVERSITÀ DEGLI STUDI FIRENZE

OpenMP

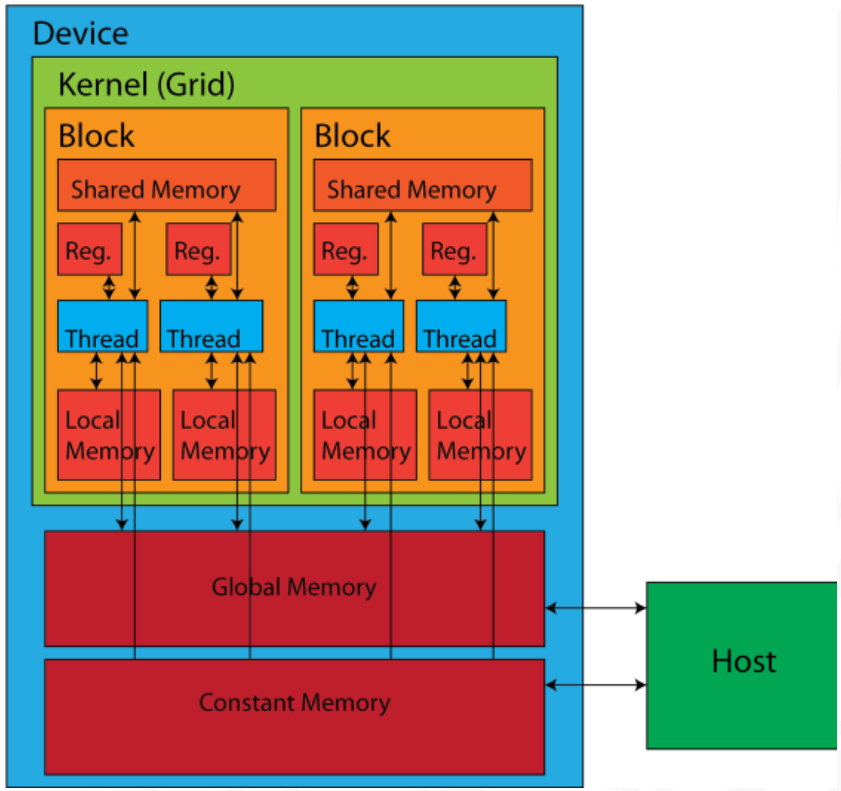
- Divisione equa dei punti da processare
- Calcolo della mode in modo indipendente:
  - nessuna necessità di comunicazione
- Riduzione effettuata in modo sequenziale
- Test effettuati sia con AoS che SoA

OpenMP

```
// ...

// compute the means
#pragma omp parallel default(none) shared(points, means, modes) firstprivate(epsilon, squaredBandwidth, nOfPoints)
{
    #pragma omp for
    for (int i = 0; i < nOfPoints; ++i)
    {
        // initialize the mean on the current point
        float mean[CHANNELS];
        for (int k = 0; k < CHANNELS; ++k) { mean[k] = points[i * CHANNELS + k]; }

        // assignment to ensure the first computation
        float shift = epsilon;

        while (shift >= epsilon)
        {
            // initialize the centroid to 0, it will accumulate points later
            float centroid[CHANNELS];
            for (int k = 0; k < CHANNELS; ++k) { centroid[k] = 0; }

            // ...
```

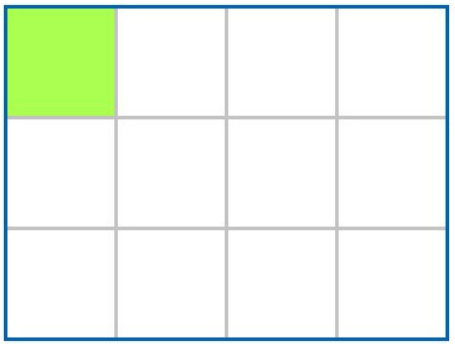
CUDA

- Architettura per parallelizzazione su GPU Nvidia
- GPU (device) vista come coprocessore della CPU (host)
- Alto numero di unità di calcolo
- Varie strutture di memoria

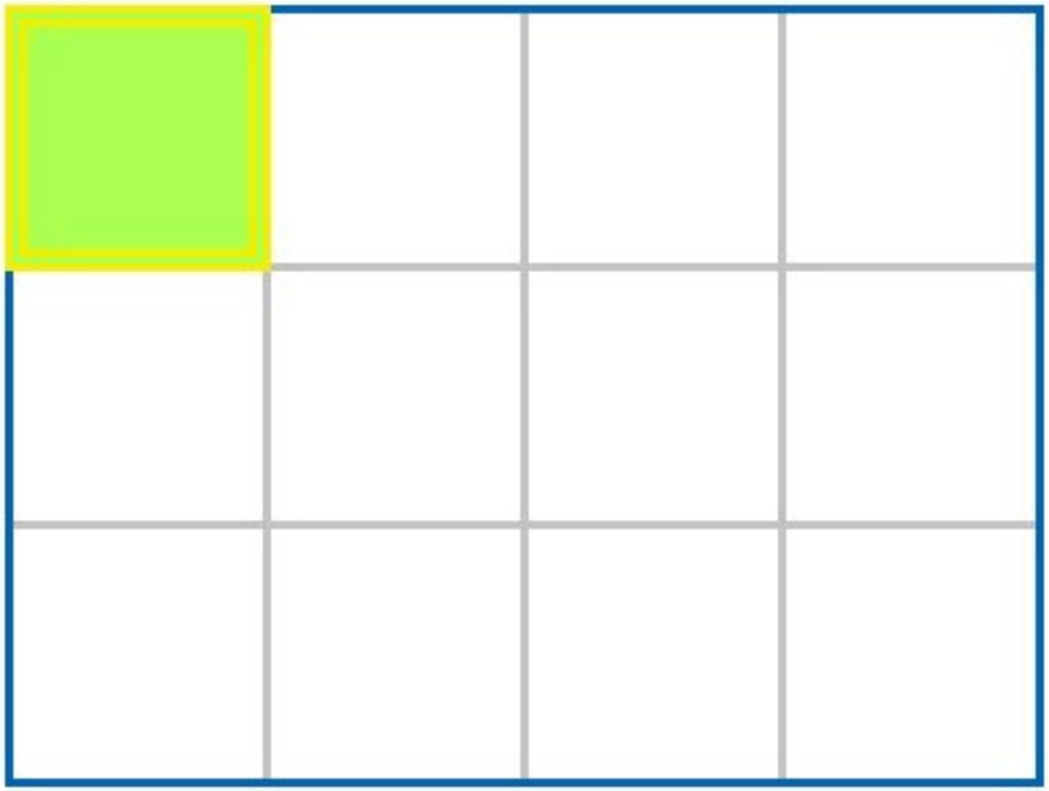


CUDA

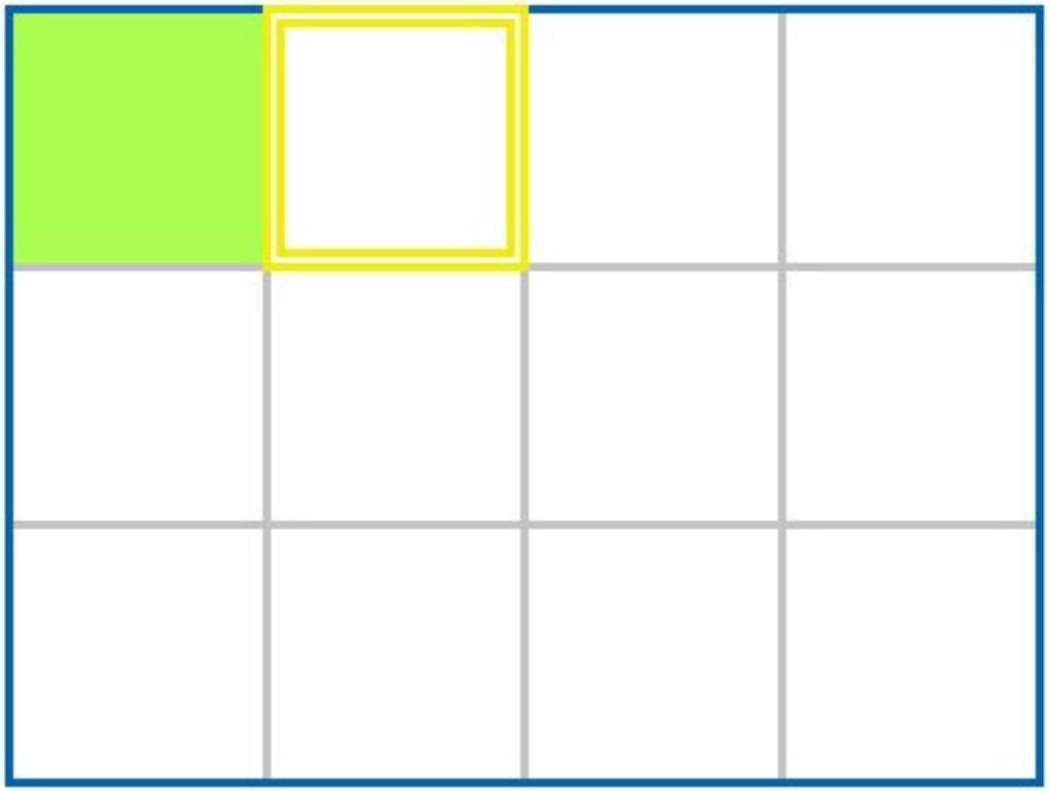
- Ogni Thread è responsabile del calcolo di una singola mode
- I Block di thread rappresentano chunk 2D dell'immagine
- Ogni punto viene letto da tutti i thread per ogni iterazione:
  - condivisione dei dati a livello di blocco,
  - strategia di Tiling per sfruttare la Shared Memory
- Riduzione effettuata all'esterno del kernel



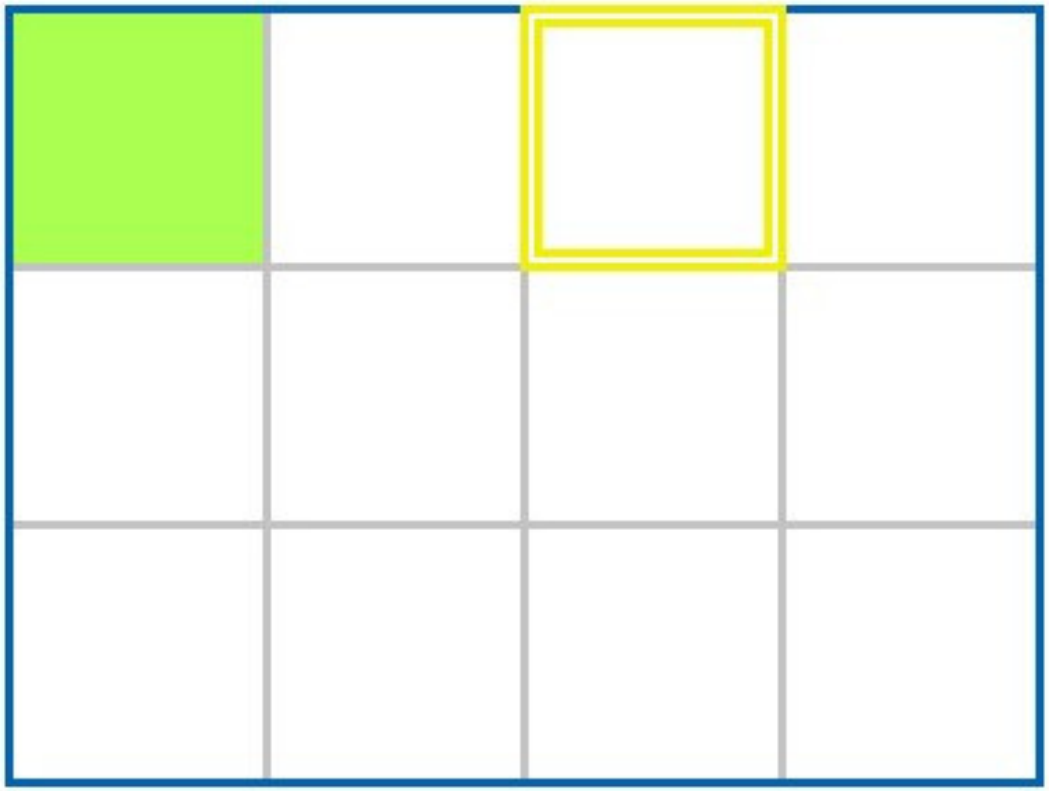
CUDA



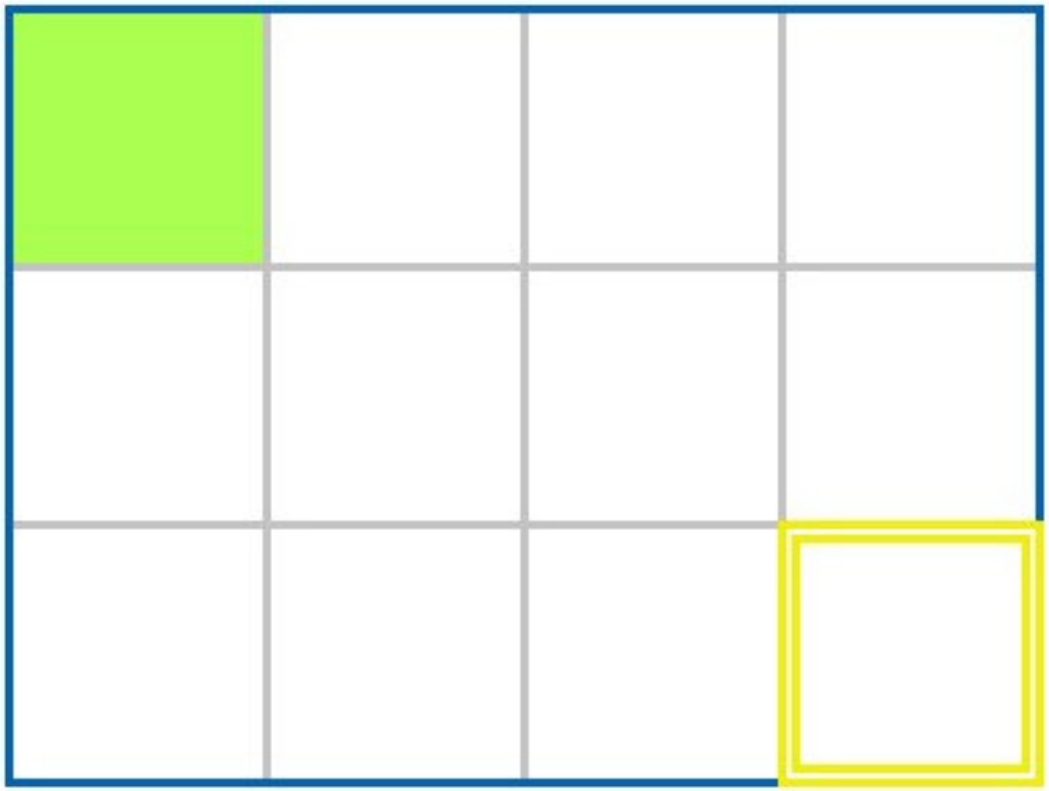
CUDA



CUDA



CUDA



CUDA

```
for (int i=0; i<loadingStepsX; i++){
    for(int j=0; j<loadingStepsY; j++){
        if (ty + THREADS_Y * j < tileDimY && tx + THREADS_X * i < tileDimX)
        {
            unsigned int phaseRow = phaseY * TILE_WIDTH + row % TILE_WIDTH + THREADS_Y * j;
            unsigned int phaseCol = phaseX * TILE_WIDTH + col % TILE_WIDTH + THREADS_X * i;
            unsigned int phasePos = (phaseRow * width + phaseCol) * CHANNELS;

            for (int k = 0; k < CHANNELS; ++k)
            { shared_tile[ty + THREADS_Y * j][(tx + THREADS_X * i) * CHANNELS + k] = points[phasePos + k]; }
        }
    }
}

__syncthreads();
```

CUDA

```
// compute the mean
if (private_continueIteration && row < height && col < width)
{
    for (int tileRow = 0; tileRow < tileDimY; ++tileRow)
    {
        for (int tileCol = 0; tileCol < tileDimX; ++tileCol)
        {
            float point[CHANNELS];
            for (int k = 0; k < CHANNELS; ++k) { point[k] = shared_tile[tileRow][tileCol * CHANNELS + k]; }

            if (l2SquaredDistance_cuda(mean, point, CHANNELS) <= const_squaredBandwidth)
            {
                // accumulate the point position
                for (int k = 0; k < CHANNELS; ++k)
                {
                    centroid[k] += point[k];
                }
                ++windowPoints;
            }
        }
    }
}

// reset shared_continueIteration
atomicAnd((int*) &shared_continueIteration, false);

__syncthreads();
```

CUDA

- Il numero di iterazioni per punto è stocastico:
  - alcuni thread potrebbero terminare prematuramente,
  - il tiling necessita che tutti i thread carichino i dati
- Introduzione di una comunicazione a livello di blocco:
  - una variabile condivisa coordina il numero di iterazioni,
  - una variabile privata stabilisce se consumare o meno i dati

```
// shared_continueIteration is true if at least one thread per block must continue
while (shared_continueIteration)
{
    float centroid[CHANNELS];

    // initialize the centroid to 0 to accumulate points later
    for (float& k : centroid) { k = 0; }

    // track the number of points inside the const_squaredBandwidth window
    int windowPoints = 0;
```

CUDA

```
// check if the thread pixel is not outside the image
if (private_continueIteration && row < height && col < width) {
    // get the centroid dividing by the number of points taken into account
    for (float& k : centroid) { k /= (float) windowPoints; }

    float shift = l2SquaredDistance_cuda(mean, centroid, CHANNELS);

    // update the mean
    for (int k = 0; k < CHANNELS; ++k) { mean[k] = centroid[k]; }

    private_continueIteration = false;

    // set if the thread must continue, hence the block
    if (shift >= epsilon) {
        atomicOr((int *) &shared_continueIteration, true);
        private_continueIteration = true;
    }

    for (int k = 0; k < CHANNELS; ++k) { means[pos + k] = mean[k]; }
}

__syncthreads();
```

Risultati

- Clustering apprezzabile
- Stesso risultato finale con entrambi i metodi
  - $\rightarrow$ Ripetibilità

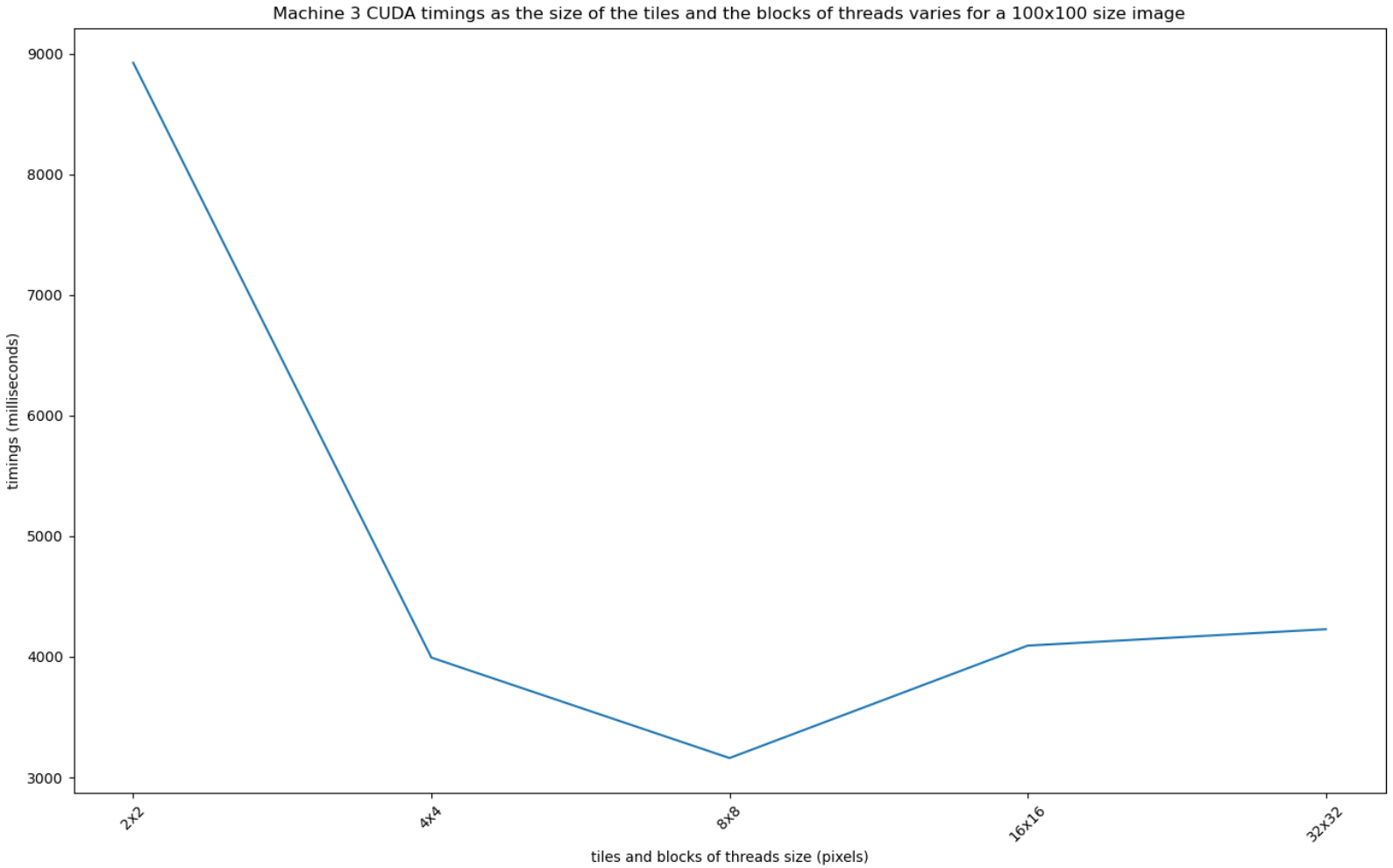
Original Output

- Performance nettamente differenti

Risultati

Machine 1					
Image dimension	Sequential AoS	Sequential SoA	OpenMP AoS	OpenMP SoA	CUDA
100 × 100 pixel	102810*ms*	102188*ms*	17047*ms*	15657*ms*	10477*ms*
250 × 250 pixel	3955082*ms*	3886222*ms*	575754*ms*	554569*ms*	276233*ms*
Machine 2					
Image dimension	Sequential AoS	Sequential SoA	OpenMP AoS	OpenMP SoA	CUDA
100 × 100 pixel	37370*ms*	35758*ms*	9937*ms*	10056*ms*	12892*ms*
250 × 250 pixel	1442048*ms*	1393829*ms*	467553*ms*	494691*ms*	355589*ms*
Machine 3					
Image dimension	Sequential AoS	Sequential SoA	OpenMP AoS	OpenMP SoA	CUDA
100 × 100 pixel	87317*ms*	86241*ms*	16547*ms*	15367*ms*	3009*ms*
250 × 250 pixel	3339803*ms*	3309131*ms*	601750*ms*	574439*ms*	56213*ms*

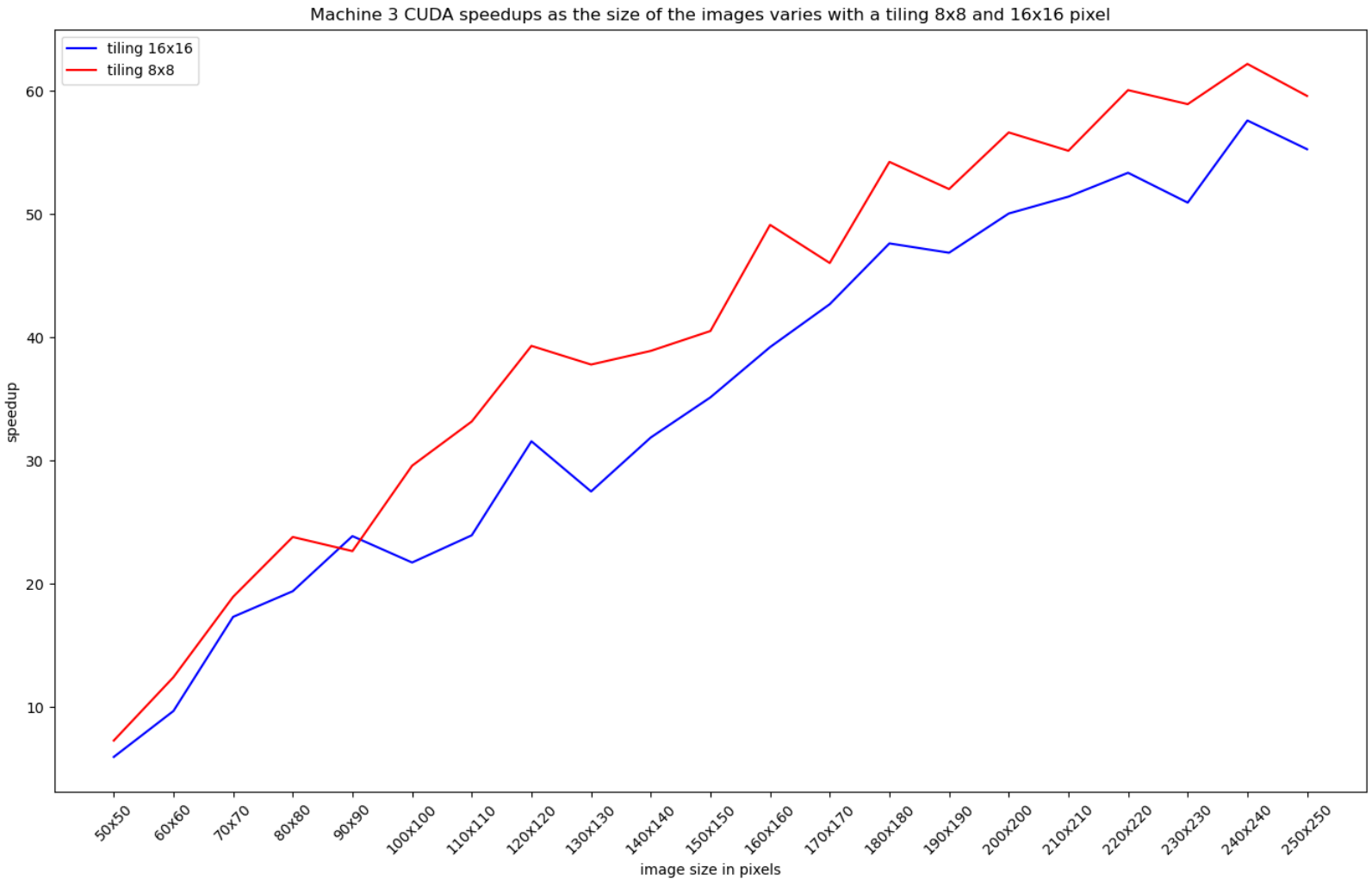
Risultati


Machine 3 CUDA timings as the size of the tiles and the blocks of threads varies for a 100x100 size image
timings (milliseconds)
9000
8000
7000
6000
5000
4000
3000
2x2
4x4
8x8
16x16
32x32
tiles and blocks of threads size (pixels)

Risultati

Machine 1			
Image dimension	OpenMP AoS	OpenMP SoA	CUDA
100 × 100 pixel	6.0	6.5	9.8
250 × 250 pixel	6.9	7.0	14
Machine 2			
Image dimension	OpenMP AoS	OpenMP SoA	CUDA
100 × 100 pixel	3.7	3.6	2.9
250 × 250 pixel	3.0	2.8	4.1
Machine 3			
Image dimension	OpenMP AoS	OpenMP SoA	CUDA
100 × 100 pixel	5.3	5.6	29.0
250 × 250 pixel	5.6	5.8	59.4

Risultati


Machine 3 CUDA speedups as the size of the images varies with a tiling 8x8 and 16x16 pixel
tiling 16x16
tiling 8x8
speedup
10
20
30
40
50
60
image size in pixels
50x50
60x60
70x70
80x80
90x90
100x100
110x110
120x120
130x130
140x140
150x150
160x160
170x170
180x180
190x190
200x200
210x210
220x220
230x230
240x240
250x250

UNIVERSITÀ DEGLI STUDI FIRENZE

Conclusioni

- OpenMP:
  - struttura SoA vantaggiosa in alcune circostanze,
  - parallelizzazione efficace con speedup sublineare
- CUDA:
  - risultati significativamente migliori,
  - speedup in crescita con l'aumento della dimensione del problema,
  - strategie come tiling e uso della shared memory sono risultate molto efficaci