Parallel Mean Shift Clustering

Elaborato Parallel Programming For Machine Learning

Nicoló Pollini, Francesco Fantechi

A.A. 2022-2023

UNIVERSITA' DEGLI STUDI DI FIRENZE
Facolta di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Contents

1 Obiettivo

L'obiettivo di questo progetto è quello di parallelizzare l'algoritmo di clustering Mean Shift tramite il framework OpenMP e l'architettura hardware CUDA e mettere a confronto i due metodi valutando gli speedup ottenuti rispetto all'esecuzione sequenziale.

2 Mean Shift

Mean Shift è un algoritmo che permette di clusterizzare un insieme N di punti appartenenti ad uno spazio delle feature a n dimensioni con un metodo equivalente all'applicazione della discesa del gradiente. L'algoritmo clusterizza i punti assegnadoli alla propria *mode*, ossia il punto a massima densitá nello spazio delle feature che gli influenza di piú. Per ogni punto viene infatti inizializzata la propria media con il proprio valore. Viene successivamente calcolato il centroide dei punti contenuti in una sfera di raggio *bandwidth* centrata nella media corrente. Questo valore appena calcolato viene poi assegnato alla media del punto e si procede ripetendo questo procedimento fino a che lo scostamento fra i due non risulti inferiore ad una certa soglia ε. Spostare la media nel centroide equivale a compiere un passo dell'algoritmo di discesa del gradiente applicato alla funzione densitá (Vedi Figura 1). Raggiunto il termine del procediemnto, la media conterrá la *mode* del punto cercata alla quale quet'ultimo verrá assegnato. Questo algoritmo risulta computazionalmente complesso ($\mathcal{O}(n^2)$) ma presenta i vantaggi di essere semplice, di non necessitare di inizializzazione, di trovare un numero di cluster arbitrario e non definito a priori e di essere robusto agli outliers. Il numero di clustering trovati e la loro dimensione dipendono dal parametro *bandwidth* scelto.

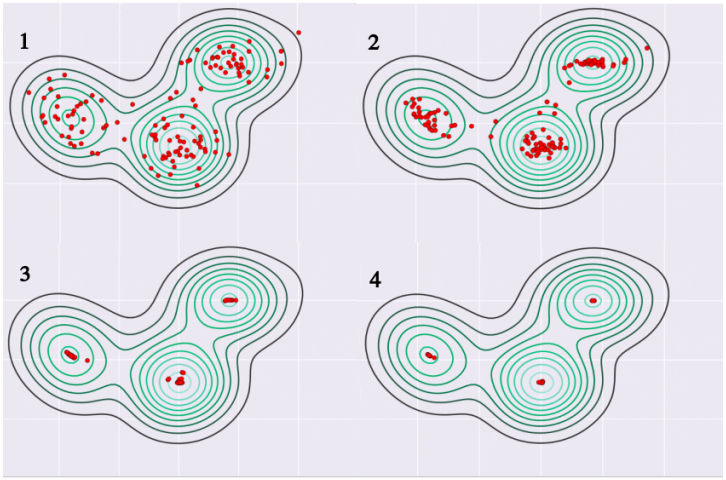

Figure 1: 2D Mean Shift simulation view

2.1 Pseudocodice

Algorithm 1 Mean Shift algorithm

```
1:  procedure MeanShift(N)  ▷ The clustering of N points in feature space
2:      shift ← ε
3:      M ← [ ]
4:      for all i in N do
5:          m_i ← i                                  ▷ Initial mean for each point
6:          while shift ≥ ε do                ▷ Stop if shift less than a threshold ε
7:              sum = 0
8:              for j ∈ N in Ball_bandwidth(m_i) do
9:                  sum ← j
10:             end for
11:             centroid ← sum/|Ball_bandwidth(m_i)|
12:             shift ← distance(m_i, centroid)
13:             m_i ← centroid
14:         end while
15:         M ← M ∪ m_i                      ▷ M contains the modes of the points
16:     end for
17:     return M
18: end procedure
```

3 Procedimento e implementazione

L'algoritmo Mean Shift é stato valutato e confrontato nella clusterizzaione di immagini in formato ".ppm" di differenti dimensioni. I nostri punti da clusterizzare saranno quindi tanti quanti i pixel che compongono l'immagine e saranno dei valori appartenenti ad uno spazio 5-dimensionale: il valore dei tre canali colore Rosso, Verde e Blu, e la posizione lungo l'asse delle ascisse e delle ordinate del pixel nell'immagine.

L'intero progetto è stato implementato in linguaggio C++ ed é stato eseguito su tre diverse macchine le cui caratteristiche sono riportate in Tabella 1. Il codice é stato versionato tramite la piattaforma GitHub ed é reperibile ai siti:
https://github.com/francesco-ftk/Parallel-Mean-Shift-OpenMP.git
https://github.com/francesco-ftk/Parallel-Mean-Shift-CUDA.git

Il progetto puó essere suddiviso in tre parti:

1. Implemenatazione dell'agoritmo Mean Shift in modo sequenziale
2. Parallelizzazione dell'algoritmo Mean Shift tramite il framework OpenMP
3. Parallelizzazione dell'algoritmo Mean Shift tramite l'architettura CUDA

3.1 Mean Shift Sequenziale

Sono state implementati due diversi modelli di design dell'algoritmo Mean Shift a seconda del layout dei dati utilizzato.
La prima versione utilizza una struttura dati a matrice per la gestione dei punti da clusterizzare. Questa tipo di struttura denominata "Array of Structures" (AoS) prevede infatti che i nostri punti 5-dimensionali siano collocati sequenzialmente in un unico vettore. Tenendo i dati localmente vicini, la struttura AoS risulta vantaggiosa per poter accedere agli elementi nel loro complesso.
La seconda versione utilizza invece una struttura dati denominata "Structure of Arrays" (SoA). Questa struttura prevede che i nostri punti 5-dimensionali siano divisi in cinque array ognuno contenente uno specifio tipo di dato. La struttura SoA per la sua forma risulta piú facilmente allineabile con la memoria cache ma necessita di un numero maggiore di accessi alla memoria per riunire i dati (Vedi Figura 2).

Entrambe le versioni sono state eseguite sull'ambiente di sviluppo CLion in modalitá "Debug" di modo da non introddure parallelizzazioni implicite da parte dell'IDE. I tempi di esecuzione in millisecondi presi su una media di dieci esecuzioni sono riportati in Tabella 2. In figura 11 é possibile visualizzare l'output ottenuto dall'algoritmo.

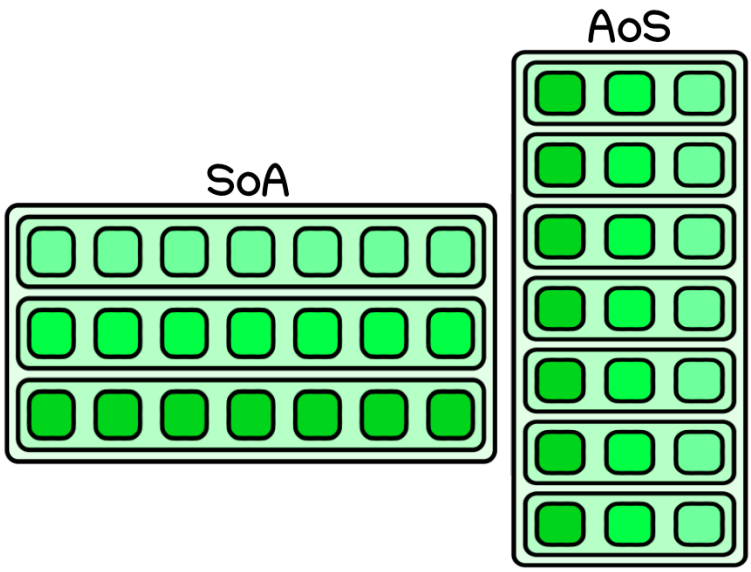

Figure 2: Data Layout: Soa vs. Aos

3.2 OpenMP

OpenMP é un framework che permette di parallelizzare una porzione di codice in modalitá implicit threading attraverso delle direttive “pragma”.
Il nostro algoritmo sequenziale puó essere diviso in due parti: la prima nella quale viene calcolata per ogni punto la propria mode attraverso l'algoritmo sopra riportato (vedi 2.1), e la seconda nella quale i punti vengono effettivamente clusterizzati assieme a seconda della vicinanza fra le loro mode. Nella prima parte i punti calcolano la loro mode in modo indipendente l'uno dagli altri. Questa sezione risuta quindi imbarazzantemente parallela e quindi facilmente parallelizzabile in quanto i thread non necessitano di comunicare o scambiare informazioni fra di loro (vedi Figura 3). La secoda parte non essendo imbarazzantemente parallela non é stata parallelizzata invece.

Il numero di thread usati da ogni macchina é stato calcolato aggiungendo un 50% al numero complessivo di Core posseduti considerando l'Hyper-Threading (vedi Tabella 1). I tempi di completamento in millisecondi presi su una media di dieci esecuzioni in modialitá “Debug” sono riportati in Tabella 2. Con questi tempi t^p e quelli presi tramite l'implementazione sequenziale t^s sono stati successivamente calcolati gli speedup delle due versioni tramite la formula $S = t^s/t^p$. Questi ultimi sono osservabili in Tabella 3.

```
// ...

// 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; }

            // ...
```

Figure 3: Piece of code parallelized via OpenMP

3.3 CUDA

CUDA é un'architettura hardware che permette di parallelizzare dei programmi general purpose eseguendo parallelamnete molti thread sulle GPU NVIDIA. La GPU (device) é vista come coprocessore della CPU (host) ed é possibile gestirne completamente la memoria. Le architetture Nvidia possiedono molteplici strutture di memoria ognuna delle quali é caratterizzata da una diversa capacitá, velocitá di accesso e condivisibilitá dei dati. Infatti, come mostrato in figura 4, queste dispongono non solo di una memoria principale (Global Memory) e di una memoria costante (Constant Memory, non modificabile dai programmi device) a cui tutti i thread possono accedere, ma anche di una memoria condivisa a livello di blocco, chiamata Shared Memory. La velocitá di accesso alla Shared Memory é significativamente piú alta rispetto alla Global Memory, tuttavia una sua limitazione é la dimensione. Non potendo contenere tutti i dati di input contemporaneamente, dopo aver portato i punti da clusterizzare da host a device, per velocizzare il loro accesso da parte dei thread dei blocchi, questi sono stati portati al momento del bisogno dalla Global Memory alla Shared Memory dei vari blocchi tramite una strategia di Tiling.

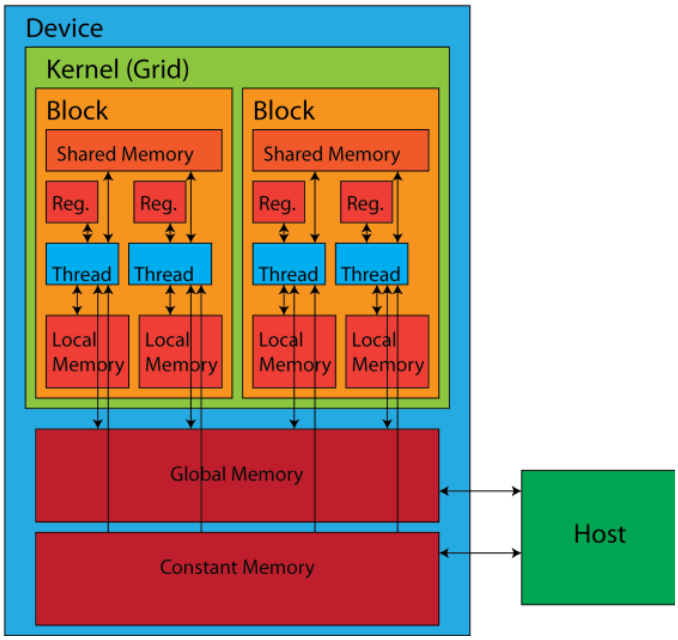

Figure 4: GPU memory view

Il Tiling, come si puó osservare in figura 5 e 6, consiste nel dividere lo spazio dei dati in piú parti e sottoporre ogni blocco di thread ad una esecuzione in piú fasi durante ognuna delle quali una sola parte dei dati viene utilizzata per la computazione. Durante le varie fasi, ogni thread è responsabile del caricamento in Shared Memory di una porzione di dati che sarà potenzialmente utilizzata da tutti i thread appartenenti allo stesso blocco, perció é cruciale che questa operazione venga eseguita tramite delle barriere di sincronizzazione.

```
for (int phaseY = 0; phaseY < phasesY; ++phaseY)
{
    for (int phaseX = 0; phaseX < phasesX; ++phaseX)
    {
```

Figure 5: Multi-phase execution due to Tiling

Figure 6: PhaseY= 0 PhaseX= 1 Tiling simulation

Nel caso del Kernel sviluppato per l'algoritmo Mean Shift, ogni thread ha la responsabilitá di calcolare la mode di un singolo pixel. Per farlo deve completare un numero indefinito di iterazioni su tutte le fasi, all'interno delle quali il pixel viene sottoposto ad una serie di operazioni che utilizzano, uno ad uno, ogni altro pixel dell'immagine. Poiché i dati di input vengono quindi utilizzati da più thread, questi sono stati portati sulla shared memory e condivisi a livello di blocco. Durante ogni iterazione, ogni blocco di thread carica in modo sincronizzato una porzione di dati in Shared Memory (vedi figura 7), la consuma (vedi figura 8), e ripete queste due operazioni per un numero di fasi sufficiente a terminare i dati complessivi.

```
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();
```

Figure 7: Loading in Shared Memory of the portion of data corresponding to the current phase

```
// 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();
```

Figure 8: Consumption of the loaded portion of data

Tuttavia, per costruzione dell'algoritmo, il numero di iterazioni che ogni thread deve eseguire per calcolare la mode del proprio pixel é variabile e non prevedibile. In altre parole, alcuni thread potrebbero necessitare di meno iterazioni di altri, e ció risulterebbe incompatibile con il caricamento sincronizzato dei dati in Shared Memory. Per gestire questo problema sono state utilizzate due variabili booleane di cui una condivisa a livello di blocco. La variabile condivisa (vedi figura 9) ha lo scopo di informare l'intero blocco di thread della necessitá di continuare con una nuova iterazione, a causa della presenza di almeno uno di loro che non ha terminato la propria esecuzione. La variabile privata invece (vedi figura 8 e 10) tiene traccia della necessitá effettiva per un thread di consumare i dati, oltre che caricarli.
Una volta che tutti i thread hanno terminato le loro iterazioni il kernel puó finalmente terminare. I dati delle mode dei pixel vengono copiati indietro sulla memoria host e l'esecuzione sequenziale per il conteggio dei cluster viene avviata.

```
// 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;
```

Figure 9: Start of a new iteration if at least one thread has not completed the mode calculation

```
// 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();
```

Figure 10: Check if mode computation is finished or another iteration is to be done

Solo la versione sequenziale AoS di Mean Shift é stata parallelizzata tramite CUDA e il suo tempo di completamento in millisecondi preso su una media di dieci esecuzioni con un tile di 8×8 pixel é riportato in Tabella 2. In Tabella 3 é possibile osservare lo speedup successivamente calcolato rispetto all'esecuzione sequenziale AoS. In figura 12 é riportato inoltre un grafico che mostra l'andamento dello speedup ottenuto dalla macchina 3 (vedi Tabella 1) al variare della dimensione dell'immagine da clusterizzare utilizzando un tiling 8×8 e 16×16, mentre in figura 13 é riporato invece un grafico che mostra i tempi di esecuzione della macchina 3 (vedi Tabella 1) al variare della dimensione del tiling su un'immagine di dimensione 100×100 pixel.

4 Risultati

Original Output

Figure 11: Mean Shift Algorithm Output

Table 1: Machine's technical specifications

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

Table 2: Timings

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

Table 3: Speedups

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

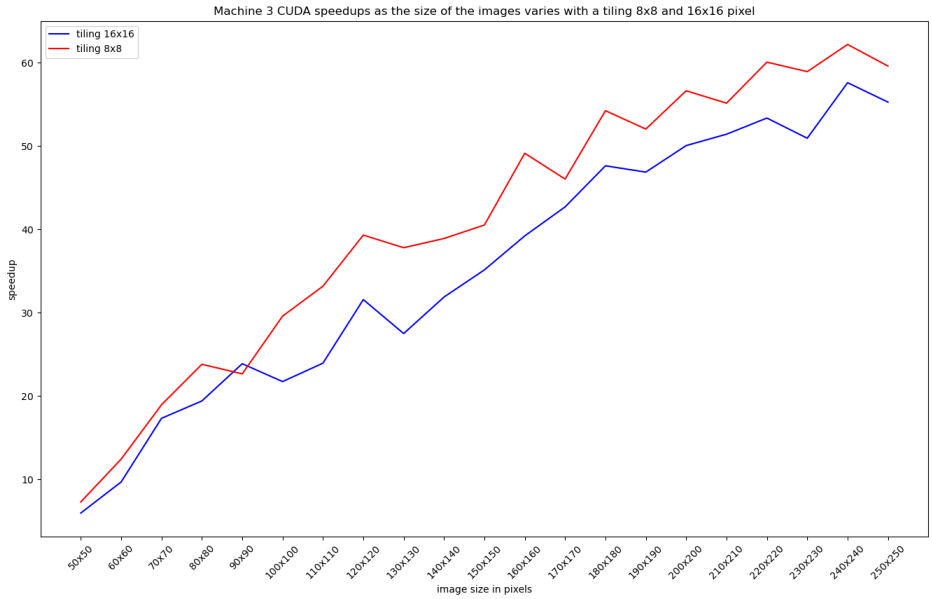

Figure 12: Machine 3 CUDA speedups as the size of the images varies

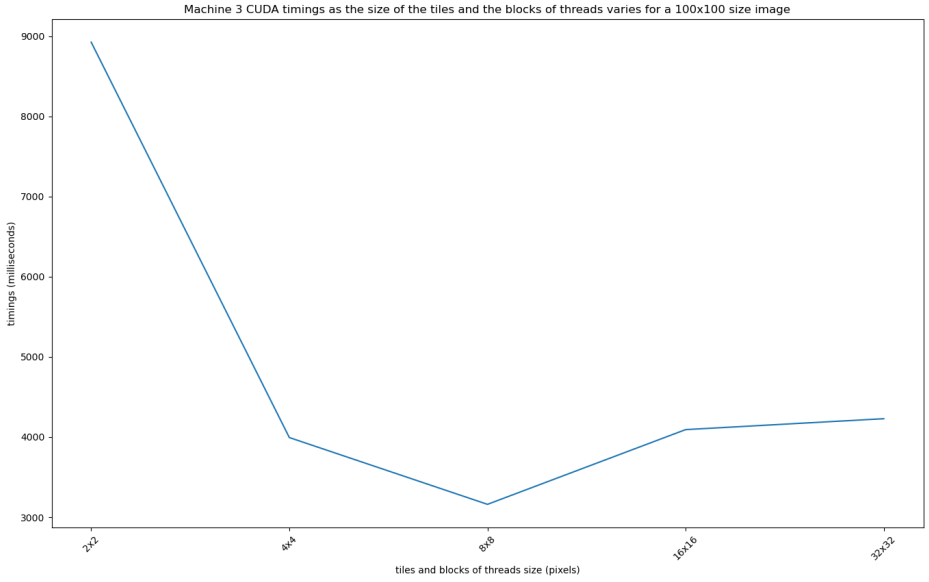

Figure 13: Machine 3 CUDA timings as the size of the tiles varies

5 Analisi e Conclusioni

Come é possibile osservare in Tabella 2 e 3, la parallelizzazione tramite OpenMP della versione dell'algoritmo Mean Shift implementata con la struttura SoA risulta piú vantaggiosa quando applicata alla macchina 1. Questo impatto é dovuto alla peculiaritá di quest'ultima di essere piú facilmente allineabile con la memoria cache; tuttavia, come si può notare dai risultati ottenuti con la macchina 2, i vantaggi di questa struttura non sono sempre apprezzabili, poiché probabilmente sensibili all'architettura hardware.
Lo speedup ottenuto risulta comunque sublineare per entrambe le versioni, mostrandosi più o meno alto a seconda della potenza specifica di ogni singola macchina.
La parallelizzazione tramite CUDA in GPU risulta, come ipotizzabile, nettamente maggiore rispetto alla parallelizzazione con OpenMP sempre proporzionalmente alle specifiche tecniche delle singole macchine. In particolare la Macchina 3 raggiunge uno speedup che, come si puó osservare nel grafico di figura 12, continua a crescere all'aumentare della dimensione dell'immagine da clusterizzare. Quest'ultimo aspetto dimostra come le architetture GPU siano piú performanti piú é alta la dimensione dei dati in ingresso, poiché riescono a sfruttare maggiormente l'elevato numero di unitá di computazione. Infine, come apprezzabile in figura 12 e 13, la dimensione dei tile e del numero di thread per blocco utilizzati impatta sui tempi di esecuzione ottenendo un minimo per la dimensione 8×8.