Parallel Computing Primo Elaborato: Analisi delle Prestazioni di Mean Shift per la segmentazione di immagini con OpenMP

Giovanni Stefanini
giovanni.stefanini@edu.unifi.it

Abstract

In questo report si analizza le prestazioni di un'implementazione del clustering Mean Shift per la segmentazione di immagini, sviluppata sia in versione sequenziale sia parallela. La versione parallela utilizza OpenMP per sfruttare il calcolo multi-threading e include ottimizzazioni con struttura dati SoA (Structure of Arrays). L'obiettivo principale è misurare lo speedup ottenuto attraverso la parallelizzazione e la vettorizzazione di Mean Shift rispetto alla sua versione sequenziale.

1. Introduzione

Mean Shift è un algoritmo di clustering non parametrico che identifica i picchi nella distribuzione di densità dei dati basandosi sulla stima della densità tramite un kernel (Kernel Density Estimation, KDE). Questo kernel definisce una finestra di ricerca, nota come bandwidth, attorno a ciascun punto per stimare la densità locale. Una delle sue caratteristiche principali è la capacità di individuare cluster senza la necessità di specificarne il numero in anticipo. L'algoritmo opera in modo iterativo, calcolando il centroide locale dei dati e spostando progressivamente i punti verso le regioni a maggiore densità. Al termine delle iterazioni, i punti che convergono verso lo stesso massimo locale vengono assegnati allo stesso cluster, con ciascun massimo che rappresenta il centro di un cluster.

Mean Shift è particolarmente efficace nella segmentazione delle immagini, poiché consente di raggruppare i pixel in base a caratteristiche simili, come il colore RGB. Durante l'iterazione dell'algoritmo, ogni pixel viene assegnato al cluster corrispondente al massimo locale individuato. Al termine del processo, l'immagine originale viene ricostruita sostituendo i pixel con i cluster identificati, formando così regioni omogenee con lo stesso colore. Questo permette di ottenere un'immagine segmentata basata sulle caratteristiche dei pixel.

Uno dei principali vantaggi di Mean Shift è che non richiede di specificare il numero di cluster in anticipo, poiché il numero di regioni viene determinato automaticamente dall'algoritmo. Inoltre, è particolarmente efficace nella segmentazione di immagini complesse, riuscendo a generare bordi netti tra le diverse regioni. Tuttavia, il suo principale svantaggio è l'elevato costo computazionale, risultando piuttosto lento su immagini di grandi dimensioni a causa della necessità di iterare su tutti i punti. Per questo motivo, una versione parallela dell'algoritmo potrebbe migliorare significativamente sia l'efficienza che i tempi di esecuzione. I **parametri principali dell'algoritmo** sono:

- **Bandwith**: ampiezza della finestra di ricerca, che determina il raggio d'azione per il calcolo del centroide.
- **Epsilon**: tolleranza per la convergenza, usata per interrompere l'iterazione quando il punto converge.

Obbiettivo principale dell'elaborato è analizzare e confrontare le performance di una implementazione sequenziale e una parallela di MeanShift.

2. Implementazione

Nell'implementazione sequenziale di Mean Shift, tutti i punti vengono analizzati uno per uno per calcolare i centroidi, mentre nella versione parallela il carico computazionale viene suddiviso tra più thread, rendendo l'elaborazione più efficiente. In entrambe le versioni, l'algoritmo è stato utilizzato per segmentare le immagini in base alla somiglianza dei colori, sfruttando esclusivamente le informazioni delle coordinate R, G e B, senza tener conto della posizione spaziale dei pixel, ovvero delle coordinate x e y.

Nel file `CMakeLists.txt` del codice è stata abilitata l'**auto-vectorization** utilizzando il comando `-ftree-vectorize`.

2.1. Versione Sequenziale

La versione sequenziale impiega una struttura dati di tipo **Array of Structures (AoS)** per rappresentare le informazioni e utilizza iterazioni semplici per determinare il centroide locale di ciascun punto. In particolare, il codice è

organizzato in due funzioni principali: `meanShift_seq`, che si occupa di calcolare i cluster a partire dai dati di input, e `segmentImage_seq`, che applica questa logica per la segmentazione di un'immagine.

Funzione `meanShift_seq`

La funzione meanShift_seq costituisce il nucleo dell'algoritmo sequenziale. Riceve in input un vettore di punti di tipo `cv::Vec3f`, dove ciascun punto rappresenta un pixel nello spazio colore RGB, gestito tramite la libreria `OpenCV`. Inoltre, accetta un parametro di banda che determina l'estensione del kernel Gaussiano e una soglia di convergenza. L'output della funzione è un vettore contenente i punti di convergenza, detti modi, che corrispondono ai cluster identificati. Il principio alla base dell'algoritmo consiste nello spostare ogni punto verso il centroide locale, calcolato in base ai punti vicini e ponderato secondo una funzione Gaussiana che assegna un peso minore ai punti più distanti. Questo processo viene ripetuto iterativamente fino a quando il movimento del punto diventa inferiore alla soglia stabilita, segnando la convergenza. Alla fine dell'elaborazione, tutti i punti si raggruppano attorno ai modi, che rappresentano i centri dei cluster individuati.

Funzione `segmentImage_seq`

La funzione segmentImage_seq utilizza Mean Shift in versione sequenziale per segmentare un'immagine. Accetta come input un'immagine in formato `cv::Mat` nello spazio colore RGB, il parametro di banda per il Mean Shift e una soglia di convergenza. Il processo inizia convertendo ogni pixel dell'immagine in un vettore RGB, definendo così lo spazio in cui verrà eseguito il clustering. Successivamente, l'algoritmo di Mean Shift viene applicato ai dati, e i risultati ottenuti vengono riorganizzati per ricostruire l'immagine segmentata. In questa nuova immagine, ogni pixel viene assegnato al colore del modo a cui è stato associato, producendo così una suddivisione in regioni cromaticamente omogenee. L'output finale è un'immagine segmentata.

Vantaggi e svantaggi versione sequenziale

Dal punto di vista computazionale, l'implementazione di Mean Shift è sequenziale, il che comporta una significativa lentezza nell'elaborazione di immagini di grandi dimensioni, a causa della complessità quadratica rispetto al numero di punti ($O(N^2)$, con N numero di pixel). Tuttavia, il codice offre un esempio chiaro e ben strutturato dell'applicazione di Mean Shift alla segmentazione delle immagini.

2.2. Versione Parallela

La versione parallela dell'algoritmo Mean Shift per la segmentazione delle immagini sfrutta **OpenMP** per distribuire i calcoli tra più thread e utilizza una rappresentazione **Structure of Arrays (SoA)** per ottimizzare l'accesso alla memoria. L'obiettivo principale è migliorare l'efficienza computazionale, permettendo di elaborare immagini di grandi dimensioni in tempi ridotti grazie al parallelismo. Il codice è strutturato attorno a quattro funzioni principali: `convertToSoA`, che trasforma i dati in formato SoA; `meanShift_parallel`, che esegue il clustering in parallelo; `reconstructFromSoA`, che riconverte i dati nel formato originale; e `segmentImage_parallel`, che applica l'algoritmo all'intera immagine per ottenere la segmentazione finale.

Funzione `convertToSoA`

La funzione `convertToSoA` trasforma i dati dell'immagine dal formato Array of Structures (AoS), in cui ogni pixel è rappresentato da un vettore RGB, al formato Structure of Arrays (SoA), dove i canali di colore (rosso, verde e blu) sono separati in array distinti. Questo approccio ottimizza la località spaziale dei dati e facilita l'elaborazione parallela.

Funzione `meanShift_parallel`

La funzione `meanShift_parallel` implementa il clustering Mean Shift in parallelo. I dati vengono inizialmente distribuiti tra i thread utilizzando una strategia di "**First Touch**", che assicura che ogni thread acceda ai blocchi di dati nella propria cache locale, riducendo così la latenza di accesso alla memoria. L'algoritmo sfrutta le direttive OpenMP per parallelizzare il ciclo principale, durante il quale ogni punto viene spostato verso il centroide locale, calcolato sulla base dei punti vicini e ponderato tramite un kernel Gaussiano. Per gestire i carichi di lavoro non uniformi tra i thread, viene utilizzato uno scheduling dinamico con blocchi di 64 elementi, garantendo un bilanciamento del carico più efficiente durante l'elaborazione. All'interno del ciclo principale, ogni thread calcola localmente i contributi cromatici dei punti vicini e aggiorna le coordinate del punto corrente in maniera indipendente. La distanza di spostamento viene utilizzata come criterio di convergenza e quando questa scende al di sotto della soglia specificata, il punto viene considerato stabile e il risultato viene salvato.

Funzione `reconstructFromSoA`

La funzione reconstructFromSoA ricostruisce l'immagine segmentata, riconvertendo i dati nel formato SoA in

un'immagine OpenCV.

Funzione `segmentImage_parallel`

Infine, la funzione segmentImage_parallel integra tutti i passaggi: converte l'immagine nel formato SoA, applica il Mean Shift in parallelo e restituisce l'immagine segmentata.

Vantaggi e svantaggi versione parallela

L'implementazione parallela di Mean Shift migliora le prestazioni rispetto alla versione sequenziale grazie all'introduzione del parallelismo con OpenMP e alla gestione efficiente della memoria tramite la struttura dati SoA. Tuttavia, il ciclo interno, che calcola il contributo dei punti vicini, rimane il principale collo di bottiglia computazionale. Quindi, i vantaggi principali sono una riduzione significativa del tempo di calcolo grazie alla parallelizzazione e all'ottimizzazione della località dei dati. D'altro canto, gli svantaggi includono una maggiore complessità nell'implementazione.

3. Misurazione delle Prestazioni

Per valutare le prestazioni, i test sono stati eseguiti su tre immagini presenti nella cartella `img` del progetto, che rappresentano un pappagallo, uno skyline e delle montagne. Tutte queste immagini hanno una risoluzione di `512x512` e sono presentate in Figura 1, ma nel codice è possibile selezionare un fattore di ridimensionamento per testare diverse risoluzioni. Le risoluzioni testate sono: `512x512`; `256x256`; `128x128`; `64x64`.

Figure 1: Immagini non segmentate selezionate per i test.

3.1. Specifiche tecniche

Il computer utilizzato per eseguire le misurazioni ha le seguenti specifiche:

Specifiche Hardware

```
CPU:Intel(R) Core(TM) i7-7700HQ Kabylake
Frequenza: 2.8 GHz
Numero di core: 4
Numero di processori logici: 8
RAM: 16 GB
GPU: NVIDIA GeForce GTX 1050
```

Specifiche Software

```
SO: Microsoft Windows 10 Home
Compilatore:  MinGW 14.2.0
IDE: CLion 2024.2.2
```

3.2. Metriche di analisi

Di seguito sono riportate le metriche impiegate per valutare e confrontare le performance delle versioni sequenziale e parallela di MeanShift.

Tempo di Esecuzione

Per misurare il tempo di esecuzione delle due versioni dell'algoritmo, è stato implementato un cronometro basato su `std::chrono` (libreria standard di C++ per la misurazione precisa del tempo). Nel codice, il tempo viene registrato solo dopo che l'utente ha scelto l'immagine da segmentare, la risoluzione da utilizzare, la versione di MeanShift da applicare e, nel caso della versione parallela, il numero di thread da impiegare. In questo modo, il tempo di esecuzione catturato riguarda esclusivamente l'esecuzione della segmentazione con MeanShift, escludendo la parte del codice precedente che serve solo a selezionare il tipo di esperimento.

Speedup

Lo Speedup (S) è una metrica che misura il miglioramento delle prestazioni di un algoritmo parallelo rispetto alla sua versione sequenziale. Esso è definito come il rapporto tra il tempo di esecuzione della versione sequenziale e quello della versione parallela:

$$\text{Speedup} = \frac{T_{\text{sequenziale}}}{T_{\text{parallelo}}}$$

dove $T_{sequenziale}$ è il tempo di esecuzione della versione sequenziale dell'algoritmo e $T_{parallelo}$ è il tempo di esecuzione della versione parallela. Quindi, lo Speedup indica quante volte l'algoritmo parallelo è più veloce di quello sequenziale. Se il numero di thread utilizzati è p, l'interpretazione che possiamo dare allo Speedup è la seguente:

- Se $S = 1 \rightarrow$ Nessun miglioramento, il parallelo è inefficace.
- Se $S > 1 \rightarrow$ Il parallelo accelera l'esecuzione.
- Se $S \approx p \rightarrow$ Speedup lineare.
- Se $S = p \rightarrow$ Speedup ideale.

- Se $S > p \rightarrow$ Speedup superlineare.

Lo Speedup è quindi un parametro essenziale per valutare quanto un algoritmo parallelo migliori le prestazioni rispetto alla versione sequenziale. Se lo Speedup aumenta significativamente con l'incremento del numero di thread, significa che l'algoritmo si presta bene alla parallelizzazione. Al contrario, se lo Speedup cresce poco o addirittura diminuisce, potrebbero esserci problemi legati all'overhead della parallelizzazione.

Efficienza del Multi-threading

L'Efficienza del multi-threading (E) è definita come:

$$\text{Efficienza} = \frac{\text{Speedup, } S}{\text{Numero di Thread, } p}$$

Se lo Speedup fosse pari al numero di thread ($S = p$), l'efficienza sarebbe del 100%, indicando che la parallelizzazione è ottimale.

4. Risultati

Le immagini risultanti dalla segmentazione con MeanShift sequenziale sono mostrate in Figura 2, mentre quelle ottenute con MeanShift parallelo sono illustrate in Figura 3. Confrontando le immagini, si nota che il risultato della segmentazione è lo stesso in entrambi i casi, il che suggerisce che entrambe le versioni stiano eseguendo correttamente il compito assegnato.

Figure 2: Immagini segmentate con algoritmo sequenziale.

Figure 3: Immagini segmentate con algoritmo parallelo.

Durante gli esperimenti, è stato osservato che per segmentare la prima immagine (pappagallo) entrambi gli algoritmi richiedevano meno tempo rispetto alle altre due immagini. Per questo motivo, per analizzare i tempi di esecuzione, lo Speedup e l'Efficienza, è stata utilizzata la prima immagine, testando diverse risoluzioni.

4.1. Analisi tempi di Esecuzione

Come anticipato, i tempi di esecuzione dell'algoritmo sequenziale e parallelo sono stati misurati sulla stessa immagine, ma con risoluzioni differenti. I risultati sono riportati nella tabella seguente:

Risoluzione	T. Sequenziale (s)	T. Parallelo (s)
512x512	2745.82	423.034
256x256	184.244	26.4912
128x128	11.373	1.55849
64x64	0.644845	0.104323

Table 1: Tempi di esecuzione in funzione della risoluzione.

Si osservi che i tempi di esecuzione della versione parallela, indicati nella Tabella 1, corrispondono al caso con 8 thread, poiché questa configurazione è la più veloce e permette di evidenziare chiaramente il vantaggio della versione parallela rispetto a quella sequenziale.

Successivamente, sono stati esaminati i tempi di esecuzione al variare del numero di thread per la prima immagine con risoluzione 128×128. Questa risoluzione è stata selezionata poiché consente di effettuare test in tempi relativamente brevi, mantenendo comunque un chiaro vantaggio della soluzione parallela rispetto a quella sequenziale. Il numero di thread utilizzati è stato variato da 1 al numero massimo disponibile sulla macchina di test (8). I risultati ottenuti sono riportati nella Tabella 2:

Numero di Threads	T. Parallelo (s)
1	9.21694
2	5.17779
3	3.36784
4	2.61946
5	2.2392
6	1.91078
7	1.69691
8	1.55849

Table 2: Tempi di esecuzione su immagini 128x128 al variare del numero di threads.

Si osserva immediatamente che la versione parallela che utilizza un solo thread risulta essere più veloce rispetto alla versione sequenziale. Questo è dovuto all'adozione della strategia "First Touch" nella versione parallela, che garantisce che ogni thread inizializzi la memoria che utilizzerà. Anche con un solo thread, questa strategia migliora la località della cache, riducendo i cache miss e ottimiz-

zando le prestazioni rispetto alla versione sequenziale. Inoltre, il compilatore, quando rileva il comando `#pragma omp parallel`, può vettorizzare il codice o applicare altre trasformazioni per migliorarne la velocità. La versione parallela utilizza anche `#pragma omp parallel for schedule(dynamic, 64) nowait` nowait, che, anche con un solo thread, suddivide il lavoro in blocchi di 64 elementi, gestendo l'esecuzione in modo più efficiente rispetto alla versione sequenziale, che esegue tutte le operazioni linearmente senza un'adeguata suddivisione del lavoro.

Infine, confrontando il tempo di esecuzione con 1 thread e quello con 8 thread, si osserva che l'utilizzo del numero massimo di thread permette di ridurre il tempo di calcolo di circa sei volte rispetto all'esecuzione con un singolo thread ($T_1 \approx 6T_8$).

4.2. Analisi dello Speedup

Proseguendo nell'analisi precedente, sono stati calcolati gli speedup in relazione alle diverse risoluzioni delle immagini e all'incremento del numero di thread. Nella Tabella 3, è presentato come varia lo speedup in base alla risoluzione dell'immagine.

Risoluzione	Speedup
512x512	6.49
256x256	6.95
128x128	7.30
64x64	6.18

Table 3: Speedup in funzione della risoluzione.

L'analisi della Tabella 3 rivela che il massimo speedup si verifica con immagini di dimensione 128x128, mentre ci si aspetterebbe un speedup maggiore per immagini di dimensione 512x512. Questo comportamento può essere spiegato dal fatto che, con risoluzioni più alte (come 512x512), la gestione della parallelizzazione può ridurre l'efficacia dello speedup. Al contrario, con risoluzioni intermedie (128x128 e 256x256), il carico computazionale è sufficientemente elevato da sfruttare in modo efficiente il parallelismo, portando a un miglioramento delle prestazioni. Invece, per immagini di dimensioni più ridotte (64×64) l'algoritmo sequenziale performa meglio, causando una diminuzione dello speedup. Tuttavia, lo speedup rimane comunque compreso tra 6.1 e 7.4, suggerendo un vantaggio consistente della versione parallela rispetto a quella sequenziale.

La Tabella 4 illustra come varia lo speedup al variare del numero di thread (da 1 a 8), mostrando lo speedup relativo. Tracciando un grafico con i dati presenti nella Tabella 4, otteniamo la Figura 4, che illustra lo speedup ottenuto in funzione del numero di thread utilizzati e lo confronta con lo speedup ideale. Dal grafico (Figura 4) si

Numero di Threads	Speedup
1	1.23
2	2.20
3	3.38
4	4.34
5	5.08
6	5.95
7	6.70
8	7.30

Table 4: Speedup Relativo su immagini 128x128 al crescere del numero di threads.

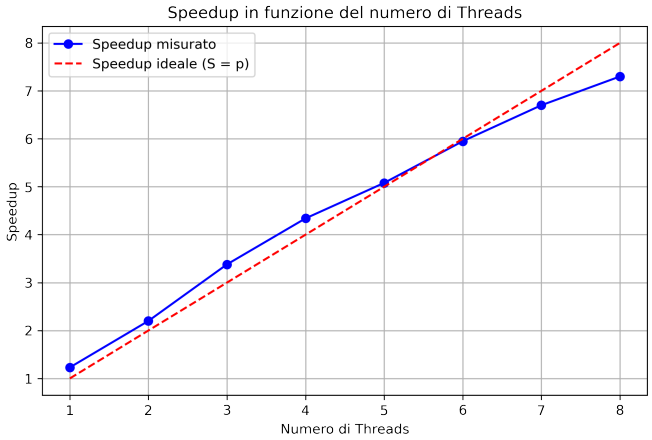

Figure 4: Speedup in funzione del numero di Threads

osserva una crescita quasi lineare ($S \approx p$) dello speedup all'aumentare del numero di thread. Pertanto, lo Speedup segue un andamento lineare, ma è probabile che, aumentando ulteriormente il numero di thread, questa curva inizi a decrescere, portando a uno Speedup sublineare. Tuttavia, con la macchina attualmente a disposizione, non è stato possibile verificare questa ipotesi.

4.3. Analisi Efficienza

Successivamente, è stata calcolata l'efficienza del multi-threading in funzione del numero di thread. I risultati sono riportati nella Tabella 5, la quale mostra che l'efficienza del multi-threading si riduce all'aumentare del numero di thread. Questo è dovuto al fatto che il sovraccarico nella gestione dei thread riduce i vantaggi derivanti dal parallelismo. Il picco di efficienza che si osserva con l'uso di un solo thread è dovuto alle ottimizzazioni aggiuntive implementate nel codice della versione parallela (come First Touch e vettorizzazione), che non sono presenti nella versione sequenziale.

Numero di Threads	Efficienza
1	123%
2	110%
3	112%
4	108%
5	101%
6	99%
7	96%
8	91%

Table 5: Efficienza su immagini 128x128 al crescere del numero di threads.

4.4. Profiling

Per eseguire il profiling del codice è stato utilizzato VTune, utilizzando come test la prima immagine a risoluzione 128x128 eseguendo Mean Shift parallelo con 8 thread. I risultati ottenuti sono mostrati in Figura 5, dove si può osservare l'attività dei 8 thread in parallelo. Inoltre, è evidente che il codice è già in esecuzione prima della sezione parallela, a causa della parte di codice in cui l'utente deve selezionare le opzioni per il test. In particolare, si possono notare 4 segmenti di attesa (indicato in bianco come "Waits") corrispondenti alle 4 scelte dell'utente (selezione dell'immagine, risoluzione, versione di Mean Shift e numero di thread).

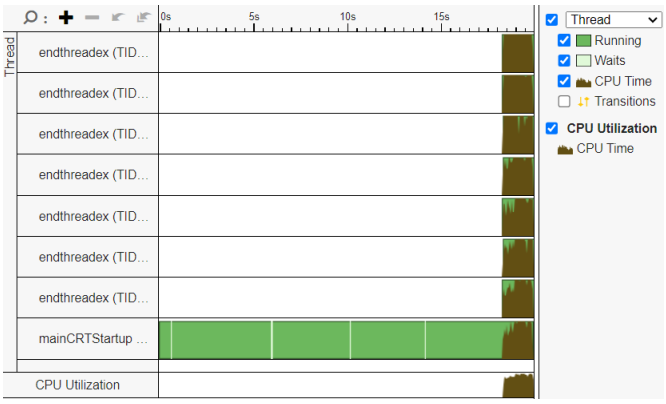

Figure 5: Profiling dei Threads con VTune

Altri risultati ottenuti con VTune sono presentati in Figura 6, in cui possiamo come VTune calcoli il tempo trascorso (elapsed time) dal momento in cui il codice inizia l'esecuzione che per la nostra analisi risulta poco interessante. Però possiamo capire quanto tempo ha impiegato il nostro algoritmo parallelo, andando a sottrarre al tempo trascorso il tempo dovuto al tempo di attesa con scarso utilizzo della CPU (Wait Time with poor CPU utilization), ottenendo 1,877s di cui i tempi dovuti all'overhead (Overhead Time) sono solo 0.052s corrispondenti al circa 3% del tempo speso in overhead invece di eseguire codice utile.

Ulteriori risultati ottenuti con VTune sono riportati in Figura 6. In questa analisi, VTune calcola il tempo totale trascorso (elapsed time) dall'inizio dell'esecuzione del codice, un'informazione di per sé poco rilevante. Tuttavia, è possibile stimare il tempo effettivo impiegato dal nostro algoritmo parallelo sottraendo dal tempo totale trascorso il tempo di attesa con scarso utilizzo della CPU (Wait Time with poor CPU utilization). In questo modo si ottiene un tempo di esecuzione pari a 1,877s, di cui solo 0,052s sono attribuibili all'overhead della parallelizzazione (Overhead Time), corrispondenti a circa il 3% del tempo totale di esecuzione. Si deduce che la parte predominante del tempo di esecuzione è stata impiegata nell'esecuzione effettiva del codice.

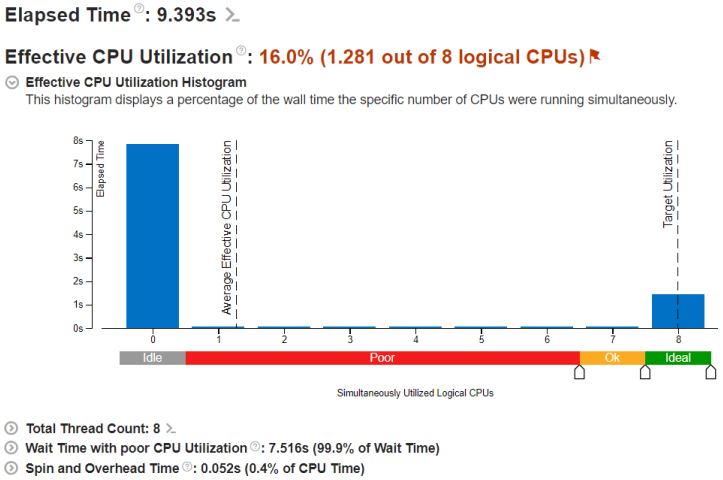

Figure 6: Profiling dei tempi con VTune

5. Conclusioni

L'implementazione parallela di Mean Shift con OpenMP mostra un notevole miglioramento delle prestazioni in termini di tempi di esecuzione rispetto alla versione sequenziale, specialmente per immagini di dimensioni maggiori. Questo miglioramento diventa più evidente aumentando il numero di thread utilizzati. Tuttavia, l'efficienza della parallelizzazione diminuisce con l'incremento del numero di thread, a causa del sovraccarico legato alla gestione dei thread.

6. Riferimenti

- Slides corso "Parallel Computing", Prof. Marco Bertini
- Documentazione OpenMP
- Image Segmentation Using Mean Shift Clustering, www.geeksforgeeks.org

Codice disponibile al seguente GitHub.