SLIC Superpixel Segmentation on CUDA

Author: Edoardo Mortara, Università degli Studi di Milano

I. Introduzione

La segmentazione nell'elaborazione digitale delle immagini è un processo di partizione in regioni di interesse di un'immagine; l'obbiettivo è ottenere una rappresentazione compatta ed estrarre oggetti o forme. In molti casi il risultato di una segmentazione può essere utilizzato per focalizzare l'attenzione di un altro sistema di visione artificiale in determinate sezioni dell'immagine. La tecnica di segmentazione SLIC (Simple Linear Iterative Clustering) Superpixels [1] viene ideata nel 2010 da Achanta e Radhakrishna, e introduce il concetto di *superpixel*.

II. Superpixels

Un superpixel è un gruppo di pixels con caratteristiche in comune; Nell'algoritmo SLIC ogni superpixel ha una posizione *x,y* e un colore RGB. Un superpixels può raggruppare a se pixels in un area quadrata limitata intorno a lui di grandezza *S*; per ogni pixel si calcolano due metriche: la prima riguarda la distanza tra la posizione del pixel e del superpixel, la seconda invece è la distanza in termini di colore.
In particolare il superpixel è un vettore $C = [r, g, b, x, y]$ detto *centro*, e la metrica utilizzata è definita come:

$$dRGB = \sqrt{(C_r - Pixel_r)^2 + (C_g - Pixel_g)^2 + (C_b - Pixel_b}$$

$$dXY = \sqrt{(C_x - Pixel_x)^2 + (C_y - Pixel_y)^2}$$

$$distance = dRGB + (m/S) + dXY$$

dove m è un coefficiente di compattezza da definire; i creatori di SLIC consigliano valori nel range $1, 20$ per lo spazio di colore CIELAB; nella mia versione ho utilizzato RGB e consiglio valori di m tra 10 e 20.
Consiglio di consultare attentamente [1] per ulteriori approfondimenti.

III. Algoritmo SLIC

La versione originale dell'algoirtmo SLIC è definita nella figura 1; questo algoritmo è molto ottimizzato rispetto ad altri algoritmi di segmentazione in quanto permette ai superpixels di analizzare solo un intorno di dimensione $2Sx2S$. Malgrado ciò, come spiegherò più avanti, la struttura di questo algoritmo non permette di ottenere grandi speedup computazionali se implementata su GPU.
Per avere prestazioni migliori occorre ridefinire l'algoritmo SLIC totalmente. Nel 2015 i ricercatori Carl Yuheng Ren, Vicotr Adrian Prisacariu e Ian D Reid hanno creato una versione di SLIC per GPU; l'algoritmo è molto diverso da quello classico: l'attenzione è spostata dai superpixels ai pixels, in modo da poter parallelizzare significativamente la computazione. L'algoritmo è riportato in figura 2.
Per entrambi gli algoritmi è necessario ripetere le computazioni dell'assegnamento e del ricalcolo dei superpixels fino a quando un errore E è minore di un certo threshold. I creatori di SLIC hanno notato che con 9/10 iterazioni si ottengono sempre risultati corretti, in cui si ha minimizzato questo errore. Nella mia implementazione ho preferito effettuare 10 iterazioni, al posto di calcolare l'errore E, per diminuire il costo computazionale.
Come mostrerò in seguito, l'algoritmo gSLICr produce risultati simili, ma non identici, all'algoritmo classico; questo è dovuto al fatto che, come indicato nella riga 9 di figura 2, ogni pixel 'osserva' 9 superpixels intorno a lui, mentre nell'algoritmo classico ogni superpixels 'osserva' un'area $2Sx2S$ di pixel nel suo intorno.

Algorithm 1 SLIC superpixel segmentation

/ Initialization */*
Initialize cluster centers $C_k = [l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regular grid steps S.
Move cluster centers to the lowest gradient position in a 3×3 neighborhood.
Set label $l(i) = -1$ for each pixel i.
Set distance $d(i) = \infty$ for each pixel i.

repeat
  / Assignment */*
  for each cluster center C_k **do**
    for each pixel i in a $2S \times 2S$ region around C_k **do**
      Compute the distance D between C_k and i.
      if $D < d(i)$ **then**
        set $d(i) = D$
        set $l(i) = k$
      end if
    end for
  end for
  / Update */*
  Compute new cluster centers.
  Compute residual error E.
until $E \leq$ threshold

Fig. 1. 'Algoritmo SLIC'

Algorithm: SLIC superpixel Segmentation

Initialize cluster centers $[l_k, a_k, b_k, x_k, y_k]^T$ by sampling pixels at regular grid steps S.
Perturb cluster centers in to the lowest gradient position.
for each pixel **do**
    Assign the pixel to the nearest cluster center based on initial grid interval S;
end for
repeat
    for each pixel **do**
        Locally search the nearby 9 cluster centers for the nearest one, then label this pixel with the nearest cluster's index.
    end for
    Update each cluster center based on pixel assignment and compute residual error E($L1$ distance) between last and current iteration.
until $E \leq threadshould$
Enforce Connectivity.

Fig. 2. 'Algoritmo gSLICr'

IV. Implementazione CUDA

Analizziamo per primo l'algoritmo SLIC su GPU; l'immagine viene suddivisa in una griglia quadrata uniforme di n colonne (da definire in input) in cui generiamo in modo regolare i nostri superpixels.
Dopo avere inizializzato i pixels (RGB, coordinate xy, label cluster e distanza) e i superpixels su CPU, copiamo tutto su GPU con *cudaMalloc* e *cudaMemcpyToSymbol* (per le costanti). La mia implementazione non utilizza ne pinned memory, ne unified memory, ma solamente global memory, per poter essere retro compatibile a CUDA Toolkit di vecchia generazione; inoltre, dopo vari test, ho notato che questa scelta non influenza significativamente le performance, infatti sarà necessario allocare poco spazio: i pixels dell'immagine e i superpixels.
Una volta eseguite 10 iterazioni sincronizzate (*cudaDeviceSynchronize*) della funzione SLIC per GPU (*slic_core*), copiamo i risultati da device a host, e li salviamo con una funzione di output chiamata *outputSlic* (vedi sezione 'Risultati').
Il kernel *slic_core* è parallelizzato rispetto ai superpixels, quindi nel caso di tanti superpixels avremo prestazioni migliori rispetto all'esecuzione su CPU. Il problema di questo kernel è la necessità di sincronizzare (*__syncthreads*) tutti i thread prima di poter assegnare a un pixel un determinato cluster (label del superpixel); questo perchè più superpixels potrebbero accedere allo stesso pixel nello stesso momento.
Se non si utilizza questa forma di sincronizzazione otteniamo risultati in cui varie regioni dell'immagine sono state assegnate in modo incorretto, o addirittura mai assegnate.

L'algoritmo gSLICr inizializza i pixels su CPU, e i superpixels su GPU; una volta trasferite tutte le informazioni utili su device iniziano le 10 iterazioni. Ogni iterazione esegue il kernel *initPixel* parallelizzato sui pixels e il kernel *slic_adv_core* parallelizzato sui superpixels. Analogamente a SLIC, bisogna utilizzare *cudaDeviceSynchronize* per attendere la fine di una iterazione prima di iniziare quella successiva. Il primo kernel si occupa di assegnare a un pixel una label (superpixel) osservando i 9 superpixels più vicini ad esso; il secondo aggiorna ogni superpixel calcolando il suo nuovo centro (posizione xy) e il suo nuovo colore RGB (media dei colori dei pixels che appartengono al superpixel).
A differenza di *slic_core* non serve nessuna sincronizzazione tra thread, quindi avremo prestazioni migliori; possiamo inoltre notare come gSLICr è decisamente più parallelizzabile rispetto a SLIC.

Come suggerito in [2], si ottengono risultati migliori con block size di 256, ma in generale qualsiasi multiplo di 64 va bene; la mia implementazione utilizza block size di 256 ($1D$) e grid size $1D$, anche se ho riscontrato che la block size influenza molto poco le performance.

V. Risultati

Dalla figura 3, alla figura 14, sono riportani alcuni risultati della segmentazione SLIC eseguita con metodo classico (GPU e CPU) e metodo gSLICr (GPU). Alcuni risultati evidenziano i bordi dei superpixels, altri invece colorano i pixels in base al colore del superpixel che li raggruppa. I centri dei superpixels sono rappresentati con piccoli pallini viola.
Come spiegato nella sezione precedente, i risultati di gSLICr sono diversi da quelli classici, malgrado ciò da un punto di vista qualitativo gSLICr produce segmentazioni più corrette; le regioni sono raggruppate con più coerenza e vengono risaltati anche piccoli particolari, ad esempio nella figura 7 sono presenti entrambe le casette bianche in centro, mentre il risultato classico ne esclude una; questa maggiore precisione può essere notata anche osservando la strada grigia che attraversa orizzontalmente l'immagine.
Un altro esempio significativo è quello della figura 13; gSLICr a parità di superpixels è riuscito a 'ricostruire' correttamente il colore dell'occhio destro, e la ciotola è più definita rispetto ai risultati classici di figura 12.
In alcuni casi SLIC e gSLICr producono segmentazioni troppo approssimate, oppure imprecise. In caso di numero troppo elevato di superpixels potremmo avere alcune regioni dell'immagine incorrettamente assegnate. Questo comportamento può essere evitato inserendo in coda all'algoritmo un'operazione di rinforzo della connettività; come spiegato in [1], questo procedimento etichetta i segmenti disgiunti con le etichette del più grande cluster vicino.
gSLICr produce risultati visivamente più corretti anche senza questa operazione finale di rinforzo.

Fig. 3. 'Esempio SLIC su GPU'

Fig. 4. 'Esempio gSLICr su GPU'

Fig. 5. 'Esempio SLIC su CPU'

Fig. 6. 'Esempio SLIC su GPU'

Fig. 7. 'Esempio gSLICr su GPU'

Fig. 8. 'Esempio SLIC su CPU'

Fig. 9. 'Esempio SLIC su GPU'

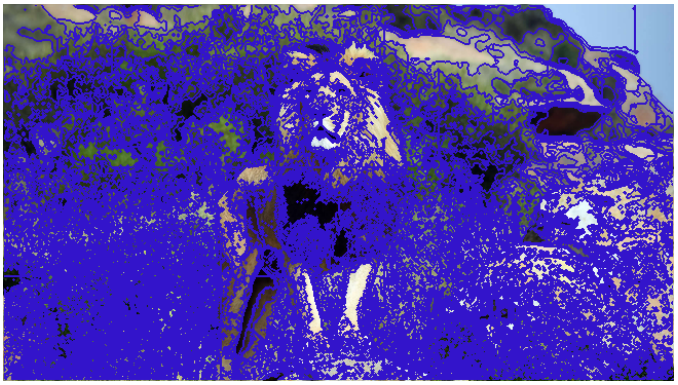

Fig. 10. 'Esempio gSLICr su GPU'

Fig. 11. 'Esempio SLIC su CPU'

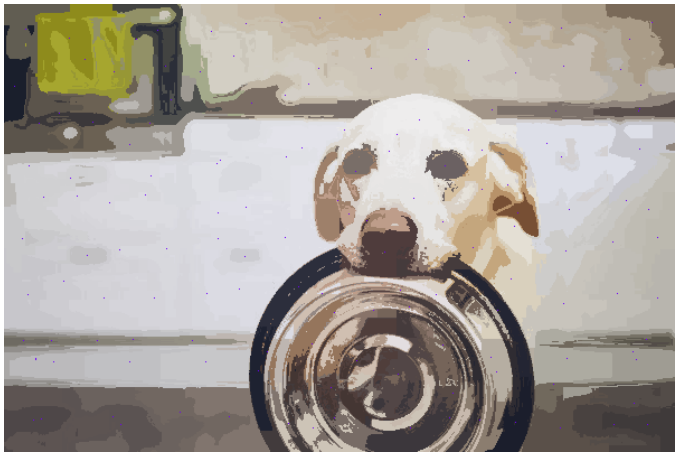

Fig. 12. 'Esempio SLIC su GPU'

Fig. 13. 'Esempio gSLICr su GPU'

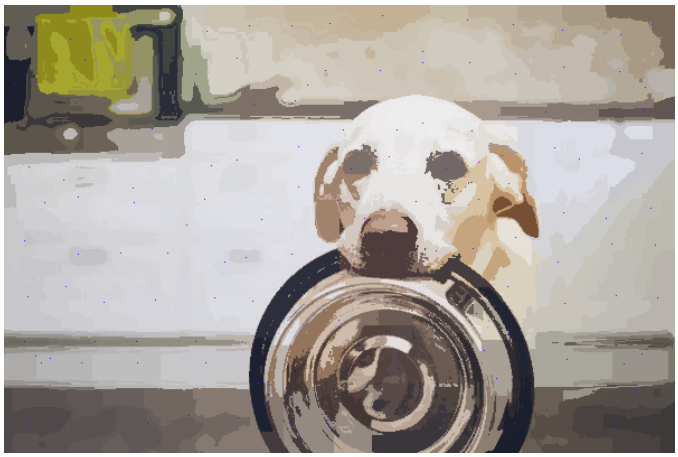

Fig. 14. 'Esempio SLIC su CPU'

VI. Profiling

La figura 15 mostra l'esecuzione di SLIC (GPU e CPU) e gSLICr su scheda Tesla M2050; l'algoritmo gSLICr permette di ottenere speedup computazionale di circa $20x$, mentre la versione SLIC su GPU ottiene solo $1.1x$.
La figura 16 mostra l'esecuzione su scheda Tesla M2090; otteniamo speed up simili a Tesla M2050 per entrambi gli algoritmi.
Utilizzando *nvprof*, come mostrato in figura 17, notiamo che l'algoritmo SLIC su GPU impiega il 95.4% del tempo di esecuzione, mentre gSLICr solo il 3.7%.
Effettuando visual profiling con *nvvp* (figura 18) possiamo osservare che le chiamate *slic_core* - azzurre a sinistra - (SLIC su GPU) impiegano molto più tempo rispetto alle chiamate *slic_adv_core* - viola a destra - (gSLICr).
L'algoritmo gSLICr risulta più veloce in quanto parallelizza la computazione in termini di pixels, mentre SLIC parallelizza in termini di superpixels. Entrambi gli algoritmo però devono attendere la computazione dei nuovi 'centri' dei superpixels prima di procedere con l'iterazione successiva; possiamo osservare questo comportamento nella figura 18 di visual profiling, infatti i segmenti colorati sono disposti in fila, uno dopo l'altro.
La figura 19 (vedi [2]) mostra esempi di tempi di elaborazione per dimensione di immagine. Per una più dettagliata analisi delle performance riferirsi a [2]; in generale con gSLICr si ottengono speedup che vanno dai $10x$ ai $83x$ (rispetto alla CPU).

```
> ./out images/dog.bmp check 30

GPUSLIC
width: 600, height: 400
N: 240000, K: 600, S: 20

GPUSLIC milliseconds: 1418.947632

GPUSLIC_ADVANCED
width: 600, height: 400
N: 240000, K: 600, S: 20

GPUSLIC_ADVANCED milliseconds: 77.715012

--------------------------------------

CPUSLIC

width: 600, height: 400
N: 240000, K: 600, S: 20

CPUSLIC milliseconds: 1579.344971

--------------------------------------

speedup CPU/GPU: 1.113040 x

speedup CPU/GPU_ADV: 20.322264 x
```

Fig. 15. 'Esecuzione su Tesla M2050'

```
> ./out images/dog.bmp check 30

GPUSLIC
width: 600, height: 400
N: 240000, K: 600, S: 20

GPUSLIC milliseconds: 1250.289795

GPUSLIC_ADVANCED
width: 600, height: 400
N: 240000, K: 600, S: 20

GPUSLIC_ADVANCED milliseconds: 65.612961

--------------------------------------

CPUSLIC

width: 600, height: 400
N: 240000, K: 600, S: 20

CPUSLIC milliseconds: 1548.038330

--------------------------------------

speedup CPU/GPU: 1.238144 x

speedup CPU/GPU_ADV: 23.593483 x
```

Fig. 16. 'Esecuzione su Tesla M2090'

```
            Type  Time(%)      Time     Calls       Avg       Min       Max  Name
 GPU activities:   95.38%  656.11ms        10  65.611ms  39.418ms  82.790ms  slic_core(center*, pixel*)
                    3.77%  25.941ms        10  2.5941ms  2.5737ms  2.6143ms  slic_adv_core(center*, pixel*)
                    0.42%  2.8594ms        11  259.95us  254.94us  269.12us  initPixel(center*, pixel*, int)
                    0.30%  2.0719ms         4  517.98us  1.6000us  1.3079ms  [CUDA memcpy DtoH]
                    0.12%  856.19us        10  85.618us     960ns  428.57us  [CUDA memcpy HtoD]
                    0.01%  56.480us         1  56.480us  56.480us  56.480us  initCluster(center*, pixel*, int)
      API calls:   73.76%  686.27ms        32  21.446ms  136.40us  82.795ms  cudaDeviceSynchronize
                   25.59%  238.06ms         6  39.677ms  6.1540us  238.02ms  cudaMemcpyToSymbol
                    0.41%  3.8320ms         8  479.00us  13.998us  1.8876ms  cudaMemcpy
                    0.10%  932.61us         1  932.61us  932.61us  932.61us  cuDeviceTotalMem
                    0.04%  392.05us        32  12.251us  4.8530us  129.51us  cudaLaunchKernel
                    0.04%  376.64us         4  94.160us  13.214us  131.73us  cudaMalloc
                    0.02%  227.74us         2  113.87us  14.303us  213.44us  cudaFree
                    0.02%  227.39us        97  2.3440us     231ns  90.983us  cuDeviceGetAttribute
                    0.00%  38.018us         6  6.3360us  3.2380us  11.237us  cudaEventRecord
                    0.00%  27.042us         1  27.042us  27.042us  27.042us  cuDeviceGetName
                    0.00%  16.777us         6  2.7960us     500ns  8.0990us  cudaEventCreate
                    0.00%  14.111us         3  4.7030us  4.3070us  5.1120us  cudaEventSynchronize
                    0.00%  10.307us         3  3.4350us  1.8690us  5.9210us  cudaEventElapsedTime
                    0.00%  4.6040us         1  4.6040us  4.6040us  4.6040us  cuDeviceGetPCIBusId
                    0.00%  4.3240us         2  2.1620us     684ns  3.6400us  cudaEventDestroy
                    0.00%  3.0800us         3  1.0260us     399ns  2.1120us  cuDeviceGetCount
                    0.00%  1.1970us         2     598ns     278ns     919ns  cuDeviceGet
                    0.00%     467ns         1     467ns     467ns     467ns  cuDeviceGetUuid
```

Fig. 17. 'nvprof'

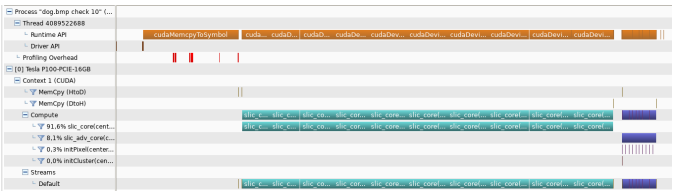

Fig. 18. 'nvvp'

Implementation	320x240	640x480	1280x960
gSLIC	9ms	21ms	86ms
SLIC [1]	88ms	354ms	1522ms

Fig. 19. 'Esempio di tempi di elaborazione completi per dimensioni dell'immagine diverse'

REFERENCES

[1] SLIC Superpixels; https://www.iro.umontreal.ca/~mignotte/IFT6150/Articles/SLIC_Superpixels.pdf.

[2] gSLICr: SLIC superpixels at over 250Hz; https://arxiv.org/abs/1509.04232; https://www.carlyuheng.com/pdfs/gSLIC_report.pdf