

Implementazione filtro di sobel in cuda c

Lorenzo Tonazzini

2019
Giugno

1 Introduzione

Il filtro di sobel è una classica trasformazione applicata a un’immagine per mettere in evidenza i contorni (edge detection problem)

L’operatore applica due kernel 3x3 di convoluzione ad un’immagine per calcolare l’approssimazione delle derivate, una in direzione orizzontale, ed una in direzione verticale.

Essendo \mathbf{Gx} , e \mathbf{Gy} i due kernel, e \mathbf{A} la matrice che rappresenta l’immagine:

1

$$\mathbf{Gx} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A} \quad \mathbf{Gy} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A} \quad (1)$$

$$\mathbf{G} = \sqrt{Gx^2 + Gy^2} \quad (2)$$

Ovviamente questa operazione va ripetuta per ogni pixel all’interno dell’immagine. Questo rende l’algoritmo **fortemente parallelizzabile**.

2 Implementazione

Essendo il programma scritto in c e volendo limitare al massimo l’utilizzo di librerie esterne, l’immagine in input viene acettata solo se in formato bitmap (con codifica di 3 byte per pixel, R, G e B).

Come prima cosa viene definita una struttura per contenere l’intestazione dell’immagine in formato bitmap, in cui si può notare come vi siano presenti all’interno tutte le informazioni riguardanti l’immagine:

```
1 typedef struct bmp_header{
2     unsigned short identifier;           // 0x0000
3     unsigned int filesize;              // 0x0002
4     unsigned int reserved;              // 0x0006
5     unsigned int bitmap_dataoffset;    // 0x000A
6     unsigned int bitmap_headersize;    // 0x000E
7     unsigned int width;                // 0x0012
8     unsigned int height;                // 0x0016
9     unsigned short planes;              // 0x001A
10    unsigned short bits_perpixel;      // 0x001C
11    unsigned int compression;            // 0x001E
12    unsigned int bitmap_datasize;       // 0x0022
13    unsigned int hresolution;           // 0x0026
14    unsigned int vresolution;           // 0x002A
15    unsigned int usedcolors;             // 0x002E
16    unsigned int importantcolors;        // 0x0032
17    unsigned int palette;               // 0x0036
18 } --attribute--((packed, aligned(1))) bmp_header; //enforce memory
                                              alignment, 1 is for not padding
```

L'immagine viene salvata in un'array di unsigned char (essendo essi di dimensione un byte) per essere convertita in scala di grigi tramite la formula:

$$\text{Gray Pixel} = 0.3 * R + 0.58 * G + 0.11 * B \quad (3)$$

Nella versione sequenziale dell'algoritmo ci si potrebbe aspettare un semplice ciclo che man mano che legge i byte dall'immagine li converte in formato gray (senza neanche salvarli in rgb).

La versione parallela è un po' più complicata, ovvero prima occorre leggere l'immagine e salvarla in un array in formato RGB, copiare i dati alla GPU ed eseguire la conversione.

Essendo un'operazione che richiede solo il pixel in esame, lanciare kernel con blocchi e griglie di thread 2D non avrebbe molto senso dal punto di vista logico, infatti si andrebbe solo a complicare notevolmente il codice.

Per ottimizzare al massimo il trasferimento di memoria si impiegano 4 stream differenti.

```

1 //pixel of image
2 unsigned char* image_data = (unsigned char*) malloc(size * 3);
3 // Input , Output gray color , Output Sobel
4 unsigned char* d_image_data , *d_gray , *d_newColors;
5
6 //Read image
7 fread(image_data , sizeof(unsigned char) , size * 3 , img);
8
9 //Memory set
10 gpuErrchk(cudaMalloc(&d_image_data , (size * 3) + size + size));
11
12 //Memory set for output data (gray image)
13 d_gray = &d_image_data[size * 3];
14 d_newColors = &d_gray[size];
15
16 int streamSize = ((size * 3) / NSTREAMS);
17 cudaStream_t streams[NSTREAMS];
18
19 int offset;
20 for (int i = 0; i < NSTREAMS; ++i) {
21     offset = i * streamSize;
22     cudaStreamCreate(&streams[i]);
23     cudaMemcpyAsync(&d_image_data[offset] , &image_data[offset] ,
24                     streamSize , cudaMemcpyHostToDevice , streams[i]);
25     cuda_gray<<<(streamSize/THREAD_IN_BLOCK_GRAY) + 1,
26                 THREAD_IN_BLOCK_GRAY , 0 , streams[i]>>>(d_image_data , offset ,
27                     streamSize , d_gray , size);
28 }
```

Alla linea 11 viene allocata tutta la memoria necessaria all'intera computazione su GPU, nelle righe successive infatti essa viene distribuita.

Notare come alla fine dell'esecuzione dei kernel non venga trasferito il risultato indietro alla CPU.

Questo perchè l'immagine in toni di grigio verrà impiegata nella computazione

successiva, applicandovi il filtro di sobel.

```
1  __constant__ float gray_value[3] = {0.3, 0.58, 0.11};
2
3  __global__ void cuda_gray(unsigned char *input, int offset, int
4      streamSize, unsigned char* gray, int size) {
5
6      int gray_idx = (offset/3) + (blockIdx.x * blockDim.x + threadIdx.x);
7      int rgb_idx = (offset) + ((blockIdx.x * blockDim.x + threadIdx.x)
8          * 3);
9
10     if (((blockIdx.x * blockDim.x + threadIdx.x)*3)>=streamSize ||
11         gray_idx>=size) {
12         return;
13     }
14
15     gray[gray_idx] = (gray_value[0] * input[rgb_idx]) + (gray_value
16         [1] * input[rgb_idx + 1]) + (gray_value[2] * input[rgb_idx +
17             2]);
18 }
```

Viene impiegata la memoria costante per limitare al massimo l'utilizzo dei registri.

Per applicare il filtro di sobel vero e proprio necessitiamo di un'altro passo.

```
1 //Set grid size
2 dim3 grid(bmp_head.width/(THREAD_IN_BLOCK*MASK) +1 , bmp_head.
3           height/(THREAD_IN_BLOCK*MASK) + 1);
4
5 //Set block size
6 dim3 block(THREAD_IN_BLOCK, THREAD_IN_BLOCK);
7
8 //Launch kernel for sobel filter
9 cuda_sobel<<<grid, block>>>(d_gray, d_newColors, bmp_head.height,
10                                bmp_head.width);
11
12 //Check for occurred error
13 gpuErrchk(cudaGetLastError());
14
15 //Copy data to host memory
16 gpuErrchk( cudaMemcpy(image_data, d_newColors, size * sizeof(
17                         unsigned char), cudaMemcpyDeviceToHost) );
```

Non è necessario sincronizzare gli stream per la conversione in toni di grigio con la funzione `cudaStreamSynchronize`, poichè essi sono bloccanti rispetto a quello di default.

Questa volta viene utilizzata una griglia di blocchi 2D poichè risulta più intuitivo per l'applicazione del filtro (essendo 3x3 e quindi anch'esso 2D).

Ora è sufficiente l'utilizzo del default stream poichè non vi è il bisogno di trasferire i dati in input (siccome sono già su GPU).

Infine viene impiegato lo stesso array inizialmente utilizzato per contenere l'immagine RGB per trasferirvi il risultato ed evitare di dover allocare ulteriore memoria.

```

1  __constant__ int sobel_x[3][3] =
2  { { 1, 0, -1 },
3  { 2, 0, -2 },
4  { 1, 0, -1 } };
5
6  __constant__ int sobel_y[3][3] =
7  { { 1, 2, 1 },
8  { 0, 0, 0 },
9  { -1, -2, -1 } };
10
11 __global__ void cuda_sobel(unsigned char* d_gray, unsigned char*
12     result, int height, int width) {
13
14     int col = blockIdx.x * blockDim.x + threadIdx.x;
15     int row = blockIdx.y * blockDim.y + threadIdx.y;
16
17     int index;
18     int gx, gy;
19     int x, y;
20
21     for(y=0; y<MASK; ++y) {
22         for(x=0; x<MASK; ++x) {
23             index = ((row * MASK) + y) * width + ((col*MASK) + x);
24
25             if(index>(width*height)) {
26                 return;
27             }
28
29             // Border Detection
30             // Bottom, top, right, left
31             if(index < ((width*height) - width) && index>(width-1) &&
32             ((index+2)%width)!=0 && ((index+1)%width)!=0) {
33                 gx = (d_gray[index - width - 1]) + (sobel_x[1][0] *
34                 d_gray[index - 1]) + (d_gray[index + width - 1]) + //1 4 7
35                 (sobel_x[0][2] * d_gray[index - width + 1]) + (sobel_x
36                 [1][2] * d_gray[index + 1]) + (sobel_x[2][2] * d_gray[index +
37                 width + 1]);
38
39                 gy = (d_gray[index - width - 1]) + (sobel_y[0][1] *
40                 d_gray[index - width]) + (sobel_y[1][0] * d_gray[index - 1]) +
41                 (d_gray[index + width + 1]) + //1 2 3
42                 (sobel_y[2][0] * d_gray[index + width - 1]) + (sobel_y
43                 [2][1] * d_gray[index + width]) + (sobel_y[2][2] * d_gray[index +
44                 width + 1]);
45
46                 result[index] = (unsigned char)min(255.0f, max(0.0f,
47                 sqrtf(gx * gx + gy * gy)));
48             }
49             else {
50                 result[index] = 0;
51             }
52         }
53     }
54 }
```

Ogni thread applicherà quindi a un quadrato di pixel di lato MASK il filtro di sobel.

La variabile index viene adibita a contenere il corrispettivo indice dell'array 1D rappresentante l'immagine, poichè non è possibile trasferire array 2D tra host e device.

Una possibile considerazione che può essere fatta è perchè non venga lanciato direttamente un solo kernel che intanto che converte in scala di grigi applichi anche il filtro di sobel.

La motivazione è principalmente poichè dovendo applicare maschere 3x3 si dovrebbe convertire varie volte lo stesso pixel in scala di grigi, aumentando di molto il numero di calcoli necessari e la complessità del problema.

Ovviamente esso è comunque risolvibile utilizzando un array di appoggio dove salvare i pixel in scala di grigi, tuttavia si andrebbe così a complicare di molto il codice, dovendo inserire meccanismi di sincronizzazione e vari condizionali, perdendo molto probabilmente il vantaggio ottenuto.

Per la gestione dei bordi si è optato per non processarli affatto, inserendo al loro posto pixel con valore 0.

Viene inoltre introdotto un meccanismo per individuare eventuali errori lato progrorammatore CUDA.

```
1 #define gpuErrchk(ans) { gpuAssert((ans), __FILE__, __LINE__); }
2
3 inline void gpuAssert(cudaError_t code, const char *file, int line,
4     bool abort=true)
5 {
6     fprintf(stderr,"GPUassert: %s %s %d\n", cudaGetErrorString(code),
7             file, line);
8     if (abort) {
9         exit(code);
10    }
11 }
```

Grazie al quale vengono identificati e stampati eventuali errori e la posizione in cui essi si sono verificati.

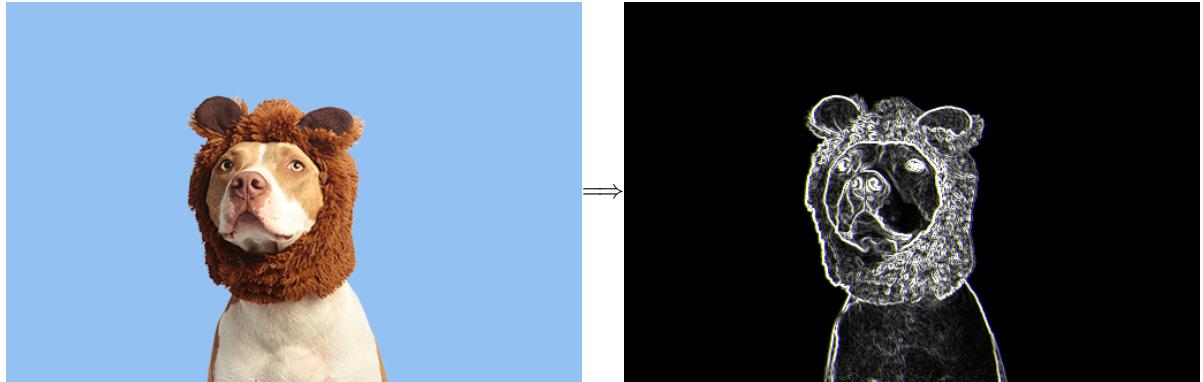
2.1 Note

Non viene impiegata la pinned memory poichè, dopo vari test, il setup per essa risulta più alto del tempo guadagnato in termini di trasferimento dati, andando a degradare le prestazioni.

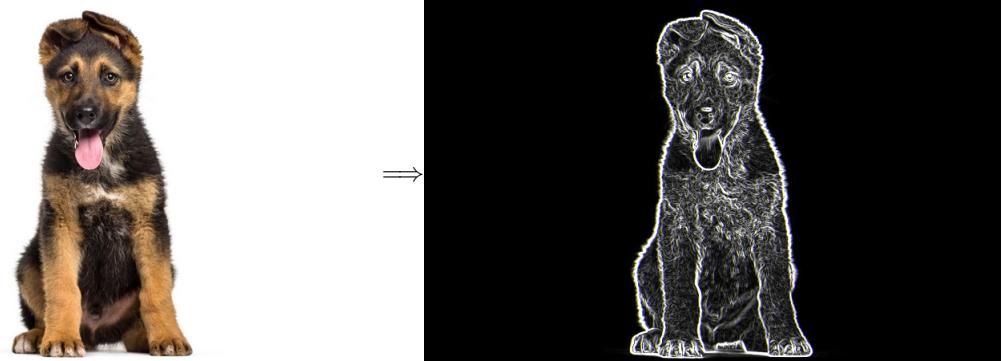
3 Test

Viene effettuato un benchmark su immagini di dimensione diversa, per testare l'algoritmo su differenti numeri di pixel.

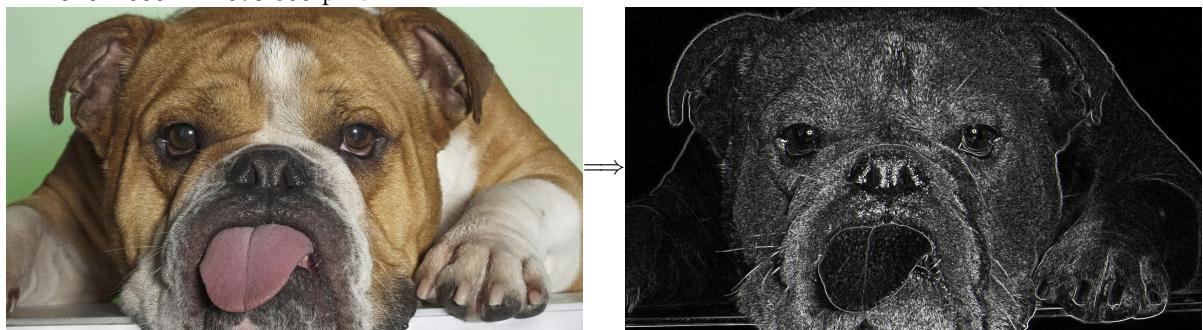
$$500 \times 333 = 166.500 \text{ pixel}$$



$$720 \times 480 = 345.600 \text{ pixel}$$



$$1920 \times 1080 = 2.073.600 \text{ pixel}$$



4 Profiling

Mettendo a confronto varie versioni dell'algoritmo risulta evidente come l'implementazione di particolari strategie possa incrementare le prestazioni.

Come ad esempio la scelta di calcolare con gpu l'immagine in scala di grigi; Si è portati a pensare che l'overhead introdotto dal dover spostare un'immagine (grande) dalla CPU alla GPU degradi le prestazioni, tuttavia ciò viene ampliamente colmato dall'estrema velocità con cui l'immagine viene convertita.

Per notare significative differenze ho dovuto testare l'algoritmo su immagini molto grandi (4096 x 3112 pixel).

L'implementazione senza GPU per il calcolo delle immagini in scala di grigio impiega mediamente **0.57 sec**, mentre nel secondo caso solamente **0.03 sec**. (calcoli effettuati sulla macchina lagrange dell'università di milano dotata di schede Tesla M2090 e Tesla M2050). Come ultima istanza è interessante mettere a confronto l'implementazione sequenziale e quella parallela, per avere un'idea del notevole speed-up ottenuto (in sec).

-	small (500x333)	medium (720x480)	big (1920x1080)	enormous(4096x3112)
seriale	0.03	0.05	0.31	1.8
parallelo	<0.01	<0.01	<0.01	0.03

Va però ricordato che ci vuole un tempo di circa 0.09 per il set-up della gpu (tempo di cui la precedente tabella non tiene conto)