di Matteo Prata

Homework 2

PARTE 1 - Memoria Globale

1. Modalità di compilazione da terminale

Per compilare il programma **blur.c** ed applicare ad ***un'immagine di input un filtro*** di una certa dimensione e salvare poi il risultato della computazione in ***un'immagine di output*** è necessario eseguire la seguente coppia di comandi:

```
$ make blur
$ ./blur [directory immagine di input] [directory immagine di output] [dimensione filtro]
```

Esempio:

```
$ make blur
$ ./blur /Users/username/Desktop/in.pgm /Users/username/Desktop/out.pgm  3
```

la sequenza di comandi nell'esempio applicherà all'immagine **in.pgm** un filtro di dimensione **3 x 3**, il risultato dell'applicazione verrà salvato nel file **out.pgm**.

2. Caratteristiche salienti dell'implementazione

Panoramica sui metodi:

Il file blur.c è strutturato nelle seguenti funzioni:

- metodo **main**
- metodo **blurer**
- metodo **clut_get_duration**
- metodo **clut_get_real_time**
- metodo **readKernelFile**
- metodo **maskCreator**

Il metodo **main** controlla che i parametri in ingresso da terminale siano corretti, nel caso non lo siano viene stampato l'errore; nel caso in cui i parametri risultino corretti vengono passati in input al metodo **blurer**.

Il metodo **clut_get_duration*** stampa la durata in secondi dell'esecuzione su GPU. Similmente il metodo **clut_get_real_time*** viene utilizzata per calcolare il tempo di esecuzione su CPU.

Il metodo **maskCreator** crea un filtro, ovvero una matrice quarta di lato dispari (valore specificato in input alla funzione) i cui elementi sono zeri e uni disposti in modo particolare. La funzione **readKernelFile** legge il contenuto di un file .cl, contenente il codice kernel OpenCL.

Il metodo **blurer** è quello che si occupa di gran parte del lavoro, esso invoca il kernel sui device e legge il contenuto del buffer di output per controllarne l'integrità e creare l'immagine di output.

** crediti a Camil Demetrescu.*

Focus sul metodo blurer:

Mediante il metodo ***blurer*** il filtro creato dal metodo **maskCreator**, idealmente, attraversa la matrice dell'immagine in input (da destra verso sinistra, dall'alto verso il basso in un contesto sequenziale).

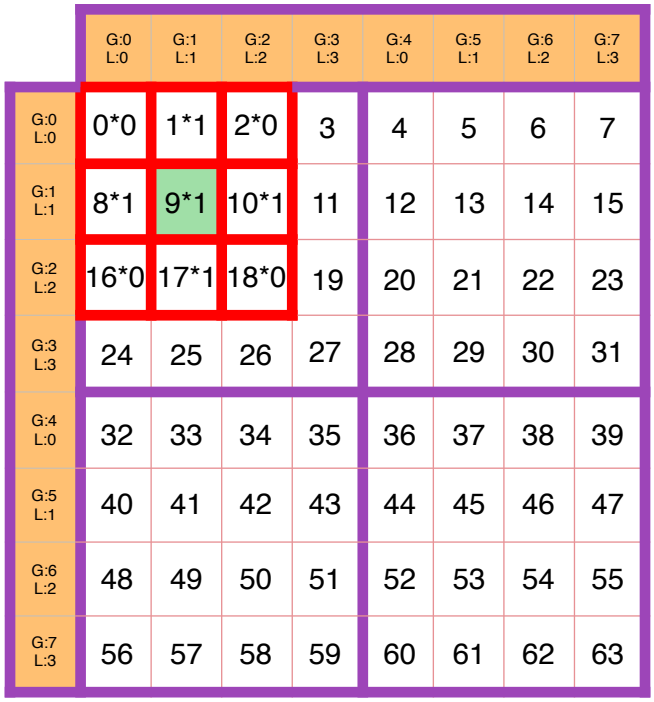
I valori dei pixel dell'immagine di input in posizione corrispondente a quelli del filtro vengono moltiplicati tra loro, la somma di questi prodotti viene divisa per il numero di uni presenti nel filtro, questo valore viene salvato in un array, che rappresenta l'immagine di output.

Se **s** è la dimensione dei lati del filtro, **h** l'altezza e **w** l'ampiezza dell'immagine di input, l'immagine di output avrà dimensione $\mathbf{d_{h,w,s}}$ = **h-2*(⌊s/2⌋)** * **w-2*(⌊s/2⌋)**. Il tempo di esecuzione dell'algoritmo è il seguente: $\mathbf{s^2}$ x $\mathbf{d_{h,w,s}}$, che corrisponde al tempo speso per effettuare le moltiplicazioni tra i valori nel filtro e quelli dell'immagine in input (s^2) per un numero di volte pari a $d_{h,w,s}$.

Esecuzione su GPU:

La memoria globale del device mantiene 3 buffer, quello che contiene i pixel dell'immagine di input, quello del filtro e quello da riempire con i pixel dell'immagine di output.

Il numero di work-group utilizzati è $\mathbf{d_{h,w,s}}$ ogni work-group si occupa di calcolare l'intensità di **uno** dei pixel dell'immagine di output. Il numero di work-items all'interno di un work-group è stato scelto sperimentalmente, **16 x 16 elementi in ogni work-group è stata reputata la scelta più efficiente**. Poiché il numero di work-items per work-group deve necessariamente essere un multiplo del numero di work-group, quest'ultimo viene opportunamente calcolato nel codice. Nel caso in cui la dimensione del work-group fosse maggiore di quella dell'immagine di input, una condizione all'inizio del codice impedisce di accedere ad indici non necessari.



L'immagine a sinistra mostra una possibile immagine in input di dimensione 8 x 8, un filtro 3 x 3, l'immagine di output $d_{h,w,s}$ = 6 x 6.

Il filtro attraversa l'immagine di input e calcola il valore dell'immagine di output. Nell'immagine viene calcolato il valore del pixel dell'immagine di output in posizione (0, 0). Il valore del pixel sarà: = (0*0 + 1*1 + 2*0 + 8*1 + 9*1 + 10*1 + 16*0 + 17*1 + 18*0)/5.

Gli indici che ottengo mediante chiamate del tipo **get_global_id** e **get_local_id** su uno specifico work-group sono indicate nella griglia affianco sulla colonna e la riga arancione. In questo esempio la local size è pari a 4 x 4 (griglie viola), essendo la dimensione dell'immagine di input un multiplo di 4 la work-group size sarà pari alla dimensione dell'immagine di input 8x 8.

3. Esempi su immagini di input

Negli esempi che seguono sono state usati filtri di dimensioni relativamente (da destra verso sinistra, dall'alto verso il basso) 1, 3, 9, 57.

4. Caratteristica della piattaforma usata per i test

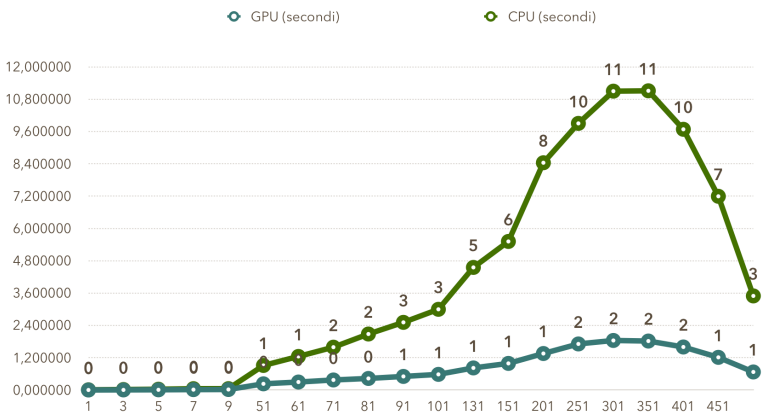
MacBook Pro 13 pollici, processore: 2, 4GHz Intel Core i5; scheda grafica: Intel Iris, 40 compute-units, clock frequency 1100 MHz, memoria globale 1536 MB, memoria locale 65536 KB, dimensione massima dei work-group 512.

5. Esperimenti cronometrati e conclusioni

L'esecuzione è evidentemente più veloce su GPU rispetto alla CPU per via del parallelismo, il valore dei pixel è calcolato parallelamente. All'aumentare della dimensione della maschera, il tempo di esecuzione sull'immagine **rectangles.pgm** (960 x 540) aumenta proporzionalmente su entrambi i device, non superando mai gli 1,8 secondi su GPU e gli 11,8 secondi sulla CPU.

Il tempo di esecuzione diminuisce drasticamente quando la maschera è grande abbastanza da diminuire il numero di computazioni necessarie a calcolare i pixel dell'immagine di output. All'aumentare della dimensione della maschera infatti diminuisce la grandezza dell'immagine di output $d_{h,w,s,}$ a prevalere invece è $\mathbf{s^2}$ che rimane comunque di dimensioni piuttosto esigue in quanto non può mai superare la grandezza dell'immagine originale.

Calcolando il rapporto tra esecuzione sequenziale su CPU ed esecuzione parallela su GPU, si è riscontrato che mediamente eseguire il programma su una GPU è **6 volte giù veloce.**



Mask	GPU (secondi)	CPU (secondi)
1	0,000302	0,006865
3	0,002517	0,017922
9	0,018081	0,049780
81	0,428496	2,079117
351	1,818068	11,112756
501	0,669589	3,494204

di Matteo Prata

PARTE 2 - Memoria Locale

La compilazione del programma **blur_local.c** è medesima alla compilazione del programma *blur.c* di cui si è parlato sopra (punto 1). Di seguito un **esempio**:

```
$ make blur_local
$ ./blur_local [directory immagine di input] [directory immagine di output] [dimensione filtro]
```

Le caratteristiche dell'implementazione del programma host sono uguali a quelle del programma *blur.c*

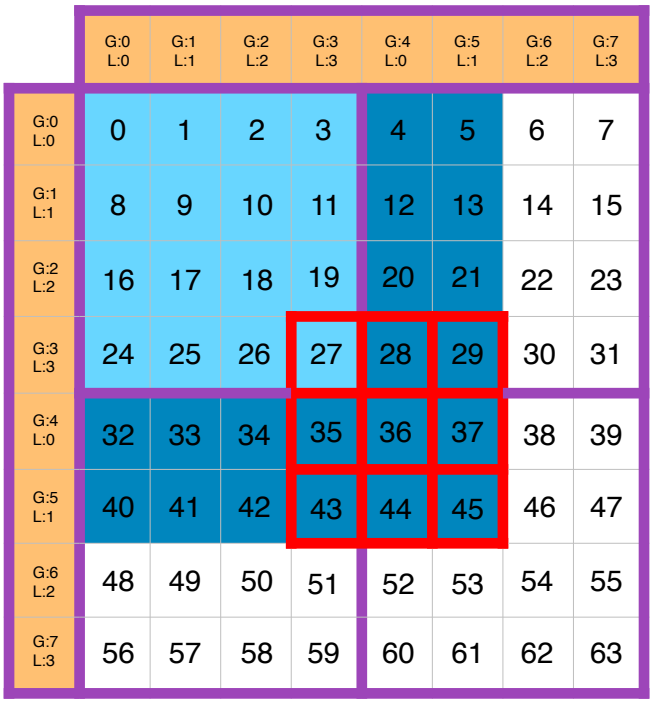
Esecuzione su GPU:

La memoria globale del device mantiene **3 buffer**, quello che contiene i pixel dell'immagine di input, quello del filtro e quello da riempire con i pixel dell'immagine di output. In questo caso, **la memoria locale** mantiene **un vettore che conterrà parte dell'immagine di input**.

Memorizzare l'immagine di input in un vettore locale, è sembrata una buona idea in quanto **accedere ad un vettore memorizzato in memoria locale è lungamente più veloce** rispetto all'accesso in memoria globale.

La struttura del Kernel (nel file **localPictureProc.cl**) che fa uso di local memory è la seguente:

1. Si ottengono gli indici globali e locali dell'attuale *item*.
2. Se gli indici globali sono all'esterno dell'immagine di input, essi vengono scartati.
3. Il **vettore** mantenuto localmente viene riempito con parte dei pixel dell'immagine di input, esattamente quelli che verranno richiesti dalla fase di elaborazione successiva.
4. Prima che venga eseguita la fase di elaborazione dei pixel, è necessario che tutti i vettori mantenuti localmente siano riempiti completamente, è necessaria quindi una **barriera** (barrier(CLK_LOCAL_MEM_FENCE)) che attenda che tutti i vettori locali vengano riempiti per poi operarvi sopra.
5. Il vettore filtro attraversa idealmente la parte di immagine memorizzata nel vettore locale, e la parte di elementi calcolati vengono man mano salvati sul vettore di immagine output.



Il valori evidenziati in azzurro (chiaro e scuro) rappresenta il contenuto della memoria locale del gruppo (0, 0) [in alto a sinistra].

I valori evidenziati in azzurro chiaro, vengono subito inseriti in memoria locale. Solo successivamente quelli in azzurro scuso (serie di istruzioni if, nel codice Kernel). I valori del bordo (azzurro scuro) vengono aggiunti per rendere possibile la computazione nella fase successiva.

Esperimenti cronometrati e conclusioni:

L'esecuzione è **più veloce su memoria locale**, a motivo del fatto che c'è **più parallelizzazione e più velocità nell'accesso alle risorse necessarie**. Il codice è ulteriormente ottimizzatile, è possibile ottimizzare la parte in cui si aggiungono elementi nella memoria locale per ottenere performance migliori ed evitare di sprecare risorse.

E' attualmente possibile avviare il programma solo con immagini le cui dimensioni sono entrambi multipli di **16**, poiché nel momento in cui mi trovassi ad escludere dei work-items con il comando che segue:

```
if (colonne >= width || righe >= height) return;
```

La **barriera** posta tra la fase di riempimento del vettore locale e la fase di elaborazione, necessità di essere eseguita su **OGNI** work-item per work-group. Una veloce modifica al codice aiuterebbe a risolvere questo problema.

Alcuni esperimenti sull'immagine **rectangles.pgm** (modificata personalmente e allegata nella cartella) hanno avuto i seguenti risultati:

Mask	GPU Local (secondi)	GPU Global (secondi)	CPU (secondi)
5	0,004414	0,004803	0,019103
7	0,008619	0,009389	0,029940
25	0,036480	0,038845	0,187618
35	0,066994	0,073536	0,355084
55	0,146269	0,150734	0,784452

L'esecuzione su memoria locale è più piuttosto simile a quella globale. Poiché il codice è ancora perfezionabile, l'esecuzione su memoria locale è migliorabile in termini di efficienza.