Kernel Image Processing in Java & Cuda

Mattia Bacci
Matricola 7060637
mattia.bacci@stud.unifi.it

Abstract

Questo elaborato verte sull'implementazione di un applicativo che effettua Kernel Image Processing sia in versione sequenziale che in versione parallela utilizzando Java e Cuda. Sono state analizzate le performance, attraverso il calcolo di speed-up ed efficienza, in un intel-i5 con 4 core per le versioni Java e la versione sequenziale in C++. Per la versione parallela attraverso CUDA i test sono stati condotti con una NVIDIA GTX 960.

1. Introduzione

1.1. Obiettivo

L'obiettivo di questo elaborato è la creazione di un applicativo sia in Java che in Cuda che effettua Kernel Image Processing su un immagine scelta. Per quanto riguarda il software Java esso è stato sviluppato partendo dall'applicazione delll'elaborato Mid-Term: "Non-blocking JPEGs Image Reader", in quanto esso permette di selezionare l'immagine sulla quale fare Kernel Image Processing.

Per questo motivo, in questa relazione non si entra nei dettagli implementativi del precedente elaborato. L'unica cosa da sapere è che dalla precedente interfaccia è possibile selezionare l'immagine da una lista.

1.2. Kernel Image Processing

La procedura consiste nel volere applicare filtri alle immagini effettuando l'operazione di convoluzione tra i pixel dell'immagine(rappresentata come una matrice bidimensionale) e una matrice detta Kernel. Tali matrici solitamente sono di piccole dimensioni(3x3, 5x5) e devono essere dispari in quanto vi è la necessita di individuare un elemento centrale.

La procedura di convoluzione consiste nell'assegnare ad un elemento la somma dei pixel vicini pesati dal kernel.

Se ne riporta la formula matematica e lo pseudocodice:

Figure 1. Esempio - Edge Detection Kernel

$$g(x,y) = \omega * f(x,y) = \sum_{dx=-a}^{a} \sum_{dy=-b}^{b} \omega(dx,dy) f(x+dx, y+dy)$$

dove *g(x,y)* è l'immagine filtrata, *f(x,y)* è l'immagine di input e ω il kernel.

```
for each image row in input image:
    for each pixel in image row:

        set accumulator to zero

        for each kernel row in kernel:
            for each element in kernel row:

                if element position  corresponding* to pixel position then
                    multiply element value  corresponding* to pixel value
                    add result to accumulator
                endif

         set output image pixel to accumulator
```

Figure 2. Pseudocodice

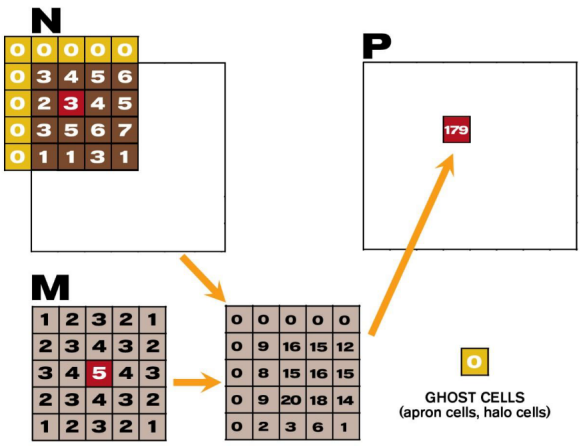

Figure 3. Convoluzione 5x5

2. Implementazione - Java

Nell'implementazione non si è fatto distinzione tra la versione sequenziale e parallela, in quella sequenziale viene eseguito solo una volta il Thread identificato dalla classe *KIPBuilder*.

2.1. I Parametri

Prima dell'inizio della computazione posso essere scelti due parametri:

1. **Numero dei Thread:** Permette di scegliere su quanti Thread effettuare la nostra esecuzione

2. **Kernel:** Permette di scegliere quale kernel applicare all'immagine.

2.2. Classi

Sono state create 4 classi principali:

1. **KIPUserInterface:** Viene inizializzata l'interfaccia grafica , è possibile selezionare cartella di output e il kernel, il numero di Thread è prelevato dall'interfaccia principale Il pulsante Salva avvia il KIPMainThread.

2. **KernelMatrix:** la matrice kernel, costituita da un array bidimensionale di interi e da un fattore moltiplicativo.

3. **KIPMainThread:** questa classe provvede alla creazione di tutti i thread KIPBuilder e attende la fine della loro esecuzione prima di salvare l'immagine. Le righe delle immagini vengono suddivise in parti uguali per i Thread, nel caso di un numero di righe non divisibili per i Thread all'ultimo vengono assegnate le righe rimanenti.

4. **KIPBuilder:** Nel costruttore della classe vengono passate le immagini di input/output, il kernel e le righe sul quale eseguire la convoluzione.

2.3. L'algoritmo

L'algoritmo opera sulle righe dell'immagine assegnate dal KIPMainThread. Per ogni Pixel i tre canali RGB sono calcolati contemporaneamente ed assegnati allo stesso tempo al pixel finale. Nel caso in cui i pixel vicini al pixel sul quale si sta eseguendo il filtraggio non esistano(casi di bordo) il loro contributo viene considerato pari a zero.

3. Implementazione - Cuda

Sono state sviluppate sia una versione sequenziale che una versione parallela in modo da poter essere confrontate tra di loro. Benchè il confronto potesse essere effettuato direttamente con la versione sequenziale scritta in Java, visto il linguaggio diverso, la diversa implementazione dell'algoritmo e della struttura dell'applicazione si è preferito scrivere una versione sequenziale in C++;

L'immagine è stata caricata in memoria utilizzando la libreria Open_CV attraverso la quale ogni immagine viene convertita in struttura a 3 canali BGR. La matrice creata posiziona i pixel dei tre canali in locazioni di memoria consecutive.

3.1. Versione Sequenziale

Per questa versione l'algoritmo, come in quello in Java, i tre canali BGR sono processati contemporaneamente e successivamente ognuno assegnato alla propria zona di memoria. Nel caso in cui i pixel vicini al pixel sul quale si sta eseguendo il filtraggio non esistano(casi di bordo) il loro contributo viene considerato pari a zero.

3.2. Versione Parallela

3.2.1 Tiling

Per ottimizzare la computazione si utilizza la strategia del Tiling in modo da minimizzare gli accessi alla memoria.

Il Tiling consiste nel caricare nella memoria di tipo Shared della GPU(molto più performante della memoria globale) sotto-parti dell'immagine denominate TILE. Nel nostro problema questa tecnica risulta molta vantaggiosa in quanto ogni pixel dell'immagine deve essere letto dalla memoria molte volte, nel caso di un kernel 3x3 ogni valore dovrebbe, al massimo, essere letto dalla memoria globale 9 volte. Con il Tiling tale valore viene copiato nella memoria shared, l'accesso alla memoria globale diminuisce da 9 a 1, gli accessi successivi a tale valore risultano molto più veloci data la differenza di banda tra le due memorie.

Java/CUDA	Risoluzione	Tempo Sequenziale	Tempo MultiThread	SpeedUp	Efficienza
Java	256x256	64 ms	21 ms	3	75%
CUDA	256x256	21 ms	690 * 10^-3 ms	~30	--%
Java	1920x1080	720 ms	210 ms	3.4	85%
CUDA	1920x1080	730 ms	17 ms	~43	--%
Java	7680×4320	10800 ms	3000 ms	3.6	90%
CUDA	7680×4320	12800 ms	260 ms	~50	--%

Figure 4. Intel Core i5 (I5-4570S) 4-Core; NVIDIA GTX 960

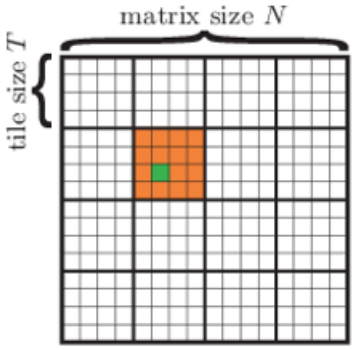

Figure 5. Tiling

Nel nostro problema vengono caricati all'interno della memoria shared non solo gli elementi appartenenti al TILE(16x16) ma anche gli elementi di bordo necessari alla computazione, per ogni TILE il numero di elementi caricati risulta uguale a $(TILE\_WIDTH + KERNEL\_WIDTH - 1)^2$. Si può calcolare il tasso di riduzione degli accessi alla memoria attraverso la formula:

$$\frac{TILE\_WIDTH^2 * KERNEL\_WIDTH^2}{(TILE\_WIDTH + KERNEL\_WIDTH - 1)^2}$$

Nel nostro caso specifico avendo un TILE_WIDTH = 16 e un Kernel con KERNEL_WIDTH = 3 il fattore di riduzione risulta uguale a 7.

Se la dimensione del KERNEL_WIDTH aumenta a 5 il fattore di riduzione risulta uguale a 16.

Da questo memento di considera $NS\_WIDTH = TILE\_WIDTH + KERNEL\_WIDTH - 1$.

3.3. Caricamento in Memoria

Per caricare i valori in memoria ci possono essere due strade:

1. Creare un Blocco con un numero di Thread uguale agli elementi del TILE(256), in questo modo i primi $NS\_WIDTH^2 - TILE\_WIDTH^2$ devono caricare un secondo elemento nella memoria Shared. **Si utilizzano tutti i Thread per la convoluzione**

2. Creare un Blocco con un numero di Thread uguale a NS_WIDTH^2, in questo caso possibile scrivere un codice più semplice ma per ogni Blocco avremmo che $NS\_WIDTH^2 - TILE\_WIDTH^2$ Thread **non effettueranno la convoluzione**.

   Nel nostro caso specifico avendo un TILE_WIDTH = 16 e un Kernel con KERNEL_WIDTH = 3 per ogni Blocco **96 Thread su 352 NON effettueranno l'operazione di convoluzione**. Si riportano i conti:

   $NS\_WIDTH = (16 + 3 - 1)^2 = 324$

   $THREADS = NS\_WIDTH + (32 - NS\_WIDTH\%32) = 352$ (arrotondati per mantenere la dimensione del Warp)

   $LOST\_THREADS = 352 - 256 = 96$

   E' stata scelta la prima modalità in modo da utilizzare tutti i Thread a disposizione.

3.3.1 L'algoritmo

L'algoritmo procede come segue:

1. L'immagine di input, output e il kernel vengono caricate nella memoria della GPU.

2. Attraverso l'utilizzo del modificatore *const* la matrice del kernel viene caricata in memoria costante non dovendo essere modificata. Si utilizza la keyword __restrict__ per permettere a GPU con Compute Capabilities >= 3.5 di utilizzare la Read-Only Cache.

3. Si dichiara una matrice all'interno della memoria condivisa che possa contenere tutti gli elementi del TILE più tutti gli elementi di bordo necessari.

4. Ogni Thread carica all'interno della memoria condivisa uno o più valori essendo tali Thread in numero uguale alla dimensione del TILE. In particolare in una prima parte tutti i Thread caricano l'elemento corrispondendo al loro ID in memoria. Nella seconda parte solamente i Thread avente ID più basso caricano in memoria gli elementi rimanenti.
5. Ogni Thread procede ad effettuare l'operazione di convoluzione per un singolo valore e il risultato viene assegnato alla matrice di output.
6. Si ripetono i punti 4 e 5 per il canale successivo, si procede dopo aver elaborato tutti e tre i canali.
7. La matrice di output viene trasferita dalla memoria della GPU alla memoria dell'host.
8. Si salva l'immagine.

4. Analisi

Per l'analisi delle performance sono state utilizzate immagini di varie dimensioni, per entrambe le versioni è riportato ed analizzato il valore di SpeedUp, per la versione in Java viene riportata anche l'efficienza. I risultati ottenuti sono una media di 5 esecuzioni arrotondate. Il kernel scelto è di tipo edge detection, la struttura della matrice è presente in Fig. 6.

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Figure 6. Edge Detection Kernel

Dalla Fig.4 possiamo trarre le seguenti conclusioni:

- **256x256** La struttura del programma Java(il dover creare 2 thread per la versione sequenziale) rallenta la computazione tanto da risultare 3 volte più lenta nel caso di piccole immagini. Il tempo sequenziale in C risulta addirittura uguale al tempo multithread in Java. Il tempo in CUDA risulta negli ordini dei microsecondi con SpeedUp pari a 30, non sono state analizzate immagini con dimensioni inferiori per non scendere sotto l'ordine dei microsecondi.
- **1920x1080** All'aumentare della risoluzione i tempi delle versioni sequenziali si stabilizzano, aumenta sia lo SpeedUp della versione in Java sia di quella CUDA.
- **7680x4320** La versione sequenziale in Java risulta più veloce di quella in C(Note ***). Aumentano gli speedup per entrambe le versioni.

4.1. Note

- La versione sequenziale del programma in CUDA è stata eseguita in una macchina locale con processore sopra citato, la versione in CUDA è stata eseguita in una macchina remota con scheda video condivisa tra più utenti e con processore ignoto.
- ***La versione sequenziale del programma in CUDA scritta in C risulta più lenta della versione sequenziale in Java, tale risultato è dovuto alla struttura della matrice Kernel nei due linguaggi. In Java è una matrice di interi, in C++ una matrice di float. Il grande numero di operazioni floating point del programma in C++ ne rallenta la computazione. In Java si evita la struttura matriciale float inserendo un fattore moltiplicativo, l'elemento da assegnare alla matrice di output è prima moltiplicato per questo valore.