UNIVERSITÀ
DEGLI STUDI
DI UDINE

Informatica Magistrale

Esame di Programmazione Parallela

Relazione Progetto
Template Matching

Luca Andaloro
Matricola 151316

Indice

1. Problema Affrontato

Il problema da affrontare è stato quello di implementare un programma che fosse in grado di riconoscere all'interno di un'immagine alcuni elementi.
Questa tipologia di problemi prende il nome di **Pattern Recognition.**
Esistono molti approcci diversi al Patter Recognition, li ho riassunti nella seguente tabella:

Approccio	Rappresentazione	Riconoscimento	Criterio di Classificazione
Basato su Modello (*Template Matching*)	Dati campionati, pixel, curve introduzione di distanze	Correlazione statistica,	Basato su stima dell'Errore di Classificazione
Classificazione Statistica (*Statistical Classification*)	Misure o Caratteristiche (Feature)	Funzioni discriminanti	Basato su stima dell'Errore di Classificazione
Sintattico o Strutturale (*Syntactic or Structural Recognition*)	Primitive	Regole, Grammatiche	Basato su stima dell'Errore di Accettazione
Reti Neurali (*Neural Network*)	Dati campionati, pixel, Caratteristiche (Feature)	Funzioni sinaptiche e di rete	Basato su stima dell'Errore quadratico medio

Per il progetto è stato scelto di utilizzare l'approccio **Template Matching.**
Ci sono due approcci al Template Matching: "Riscontro basato sul modello" (template base matching) o sulla "caratteristica" (feature based matching). Il primo usa le immagini intere sommando/comparando varie metriche (SAD, SSD, correlazione incrociata, ...) per determinare la possibile miglior posizione; il secondo usa una caratteristica della immagine-modello come per esempio i contorni o angoli come prima misura per trovare il miglior riscontro locale nell'immagine-sorgente.
L'idea di base è quella di prendere in input due immagini, la prima sarà l'immagine che noi chiameremo Origine, la seconda sarà un frammento della prima che noi chiameremo appunto Template, andremo quindi a ricercare all'interno dell'immagine origine il nostro template. Siccome non sappiamo a priori le regioni in cui il template può presentarsi, è necessario confrontarlo con tutte le sottoparti dell'immagine origine che hanno le sue stesse dimensioni. Durante il confronto viene calcolata la differenza tra i pixel del template e dell'immagine origine utilizzando la tecnica SAD (Sum of absolute differences) che fa parte dell'approccio al Template Matching chiamato "Riscontro basato sul modello":

$$SAD\,(i,j) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} |I\,(i+m, j+n) - T\,(m,n)|$$

Una volta calcolate tutte le differenze viene presa la differenza minore e quella sarà la nostra corrispondenza.

Vediamo un possibile codice sequenziale:

```
// Sposta il template sull'immagine origine e calcola la differenza tra ogni
pixels
    for (i = 0; i <= O_cols - T_cols; i++) {
        for (j = 0; j <= O_rows - T_rows; j++) {
            for (k = 0; k < T_cols; k++) {
                for (l = 0; l < T_rows; l++) {
                    pixelOrigine = O[((l + j) + (k + i)* O_rows)];
                    pixelTemplate = T[l + k*T_rows]
                    valDiff = pixelOrigine - pixelTemplate;
                    diff += ValoreAssoluto(valDiff);
                }
            }
 differences[j + i * (O_rows - T_rows + 1)] = (diff /((float)(T_cols*T_rows)));
            diff = 0;
            // Cerca la differenza migliore e prendiamo le coordinate x
                & y che corrisponderanno all'angolo alto sinistro del nostro
                rettangolo
            if ( minSAD > differences[j + i * (Iw - Tw + 1)] ) {
                minSAD = differences[j + i * (Iw - Tw + 1)];
                *x = j;
                *y = i;
            }
        }
    }
```

Trovata la differenza minore andremo a tracciare sull'immagine origine un quadrato rosso in corrispondenza del punto trovato.
Come si può notare è un codice composto da molti cicli *for* annidati, questo comporta ad un ovvio rallentamento di esecuzione ma d'altro canto, come vedremo in seguito, è un algoritmo che si presta alla paralellizzazione

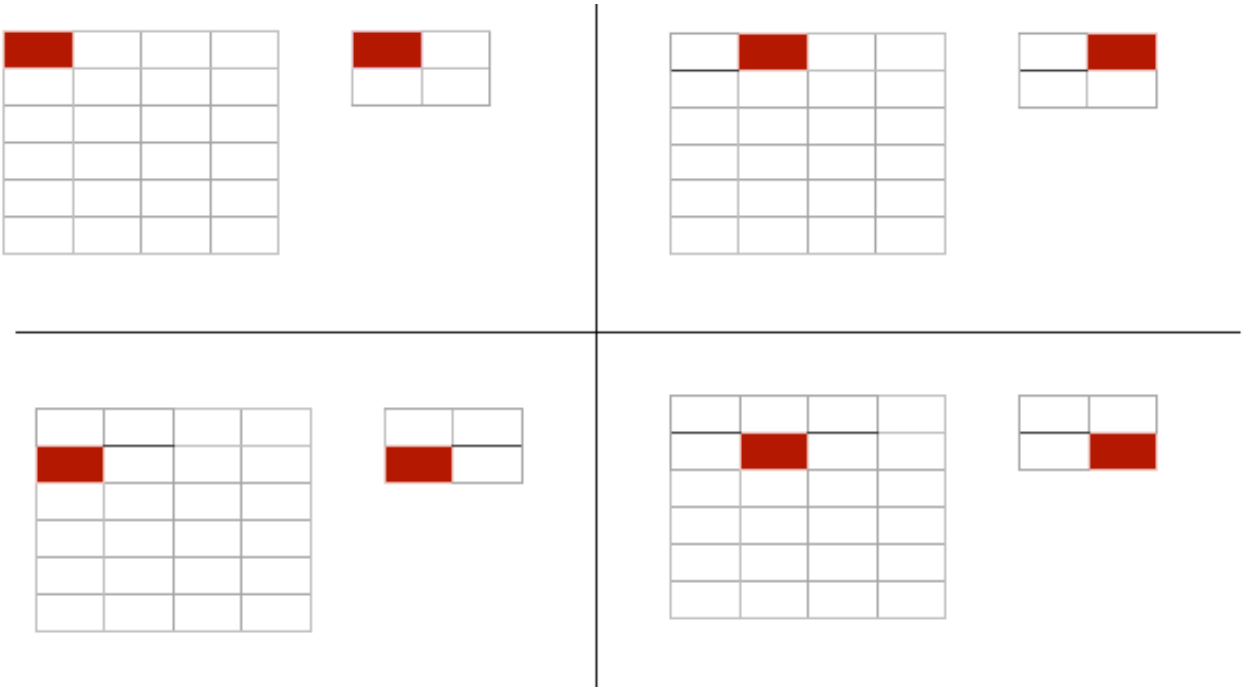
2. Metodologie adottate

2.1 Gestione immagini

Per l'immagini da utilizzare nel progetto è stato scelto il formato BMP, ricercando online ho trovato una libraria "bmp_util.c" che legge l'immagine a colori la converte in scala di grigi e inserisce tutti i valori all'interno di un array row-major. Per l'output, invece, la libreria contiene una funzione che tramite le coordinate disegna sull'immagine un quadrato rosso. Tutte queste operazioni avvengono in seriale.

2.2 Calcolo differenze

L'idea implementata per l'approccio parallelo è stata quella di far svolgere ad un singolo threads la sommatoria delle differenze per una occorrenza del template sull'immagine origine.



2.1 Esempio grafico del lavoro svolto da un singolo thread

Tramite 4 cicli for annidati facciamo scorrere l'immagine origine e l'immagine template. I due cicli più esterni sono parallelizzati tramite la tecnica gride-stride loop e ogni thread esegue i due cicli interni spostano l'immagine template sull'immagine origine e calcolando le singolo differenze tra i pixel.
Una volta deciso cosa dovesse svolgere il singolo thread si è passati a ragionare sul numero di threads e blocchi da lanciare. Per il numero di thread per blocco ho

deciso di sfruttare tutti i 1024 thread messi a disposizione dalla scheda in 2D quindi (32, 32), invece per il numero di blocchi ne vengono lanciati tanti quanti bastano a coprire ogni differenza con appunto un singolo threads. Effettuando i seguenti calcoli:

```
differenceW = (origine_width - template_width + 1);
differenceH = (origine_height - template_height + 1);
numBlocks = (differenceW * differenceH + maxThreads - 1) / maxThreads;
```

Essendo che l'accesso al template è molto frequente e si ripete molte volte durante l'esecuzione del programma si è deciso di spostarlo all'interno della memoria shared. Per fare questo ho controllato che non si verifichino dei "Bank conflict" durante l'accesso. Una vota appurato che il pattern di accesso non producesse dei conflitti è stato implementato il trasferimento del template in shared, ogni threads del blocco copia più parti del template, in base alla sua grandezza, utilizzando la tecnica grid-stride loops. La shared memory viene allocata dinamicamente in base alla dimensione del template, un controllo durante l'apertura dell'immagine avverte se il template è più grande o meno della shared Memory e nel caso termina il programma.
Un problema che si è cercato di risolvere ma purtroppo con esito negativo è il verificarsi di coalescenza durante la lettura dell'immagine origine all'interno della global Memory perchè come si vede anche dalla rappresentazione grafica precedente (2.1) l'accesso non è lineare ma segue la conformazione del template.
L'idea che si voleva applicare era di spostare anche l'immagine origine in memoria shared ma ovviamente sarebbe stato una limitazione per le sue dimensioni quindi ho pensato di caricare di volta in volta solo la parte di immagine origine da confrontare ma facendo alcune prove le prestazioni non sono migliorate forse perchè si effettua troppe volte il caricamento di parti d'immagini in shared che vengono lette solo una volta, quindi si andava a eliminare la coalescenza ma si introducevano troppi spostamenti senza portare un effettivo guadagno di prestazioni.

2.3 Ricerca del minimo

Premessa: La ricerca è stata implementata solamente per fini didattici perchè ci si è accorti fin da subito che le prestazioni del programma non miglioravano di molto, anzi era no praticamente uguali all'utilizzo della CPU, questo perchè l'array in cui andiamo a ricercare il minimo è di piccole dimensioni (comunque in fase di test è stato provato il codice di ricerca su un array di dimensioni molto più elevate e in quel caso la differenze è stata notevole, vedremo dopo nei grafici).

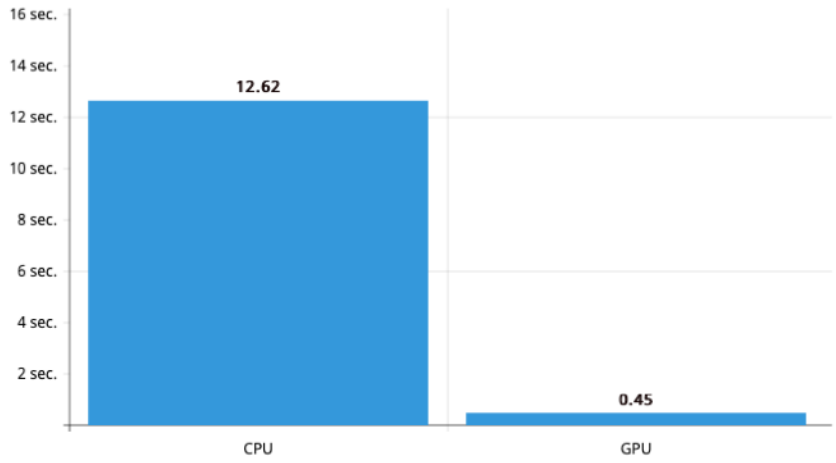
Per ricercare il valore del minimo all'interno della matrice delle differenze sfruttando il parallelismo Cuda ho preso spunto dal seguente articolo[1], dove è stato ottimizzato il ciclo di scansione dell'array assegnando a più thread la ricerca. Ma per fare questo si ha bisogno di un'operazione atomica per salvare il valore minimo trovato. Cuda mette a disposizione la funzione "atomicMin()" ma accetta sono valori interi e noi stiamo lavorando con dei float, quindi ci viene in aiuto l'articolo citato in precedenza dove viene implementata una funzione atomica per i float.
Inoltre evitiamo la copia dell'array differences da Device a host risparmiando tempo molto variabile, perchè le dimensioni di differences dipendono dell'immagine origine, invece, in questo caso ritorniamo solo gli indici che corrisponderanno all'angolo in alto sinistro del quadrato rosso sull'immagine di output.

[1] Oleksii Kupriienko - https://www.apriorit.com/dev-blog/614-cpp-cuda-accelerate-algorithm-cpu-gpu

3. Risultati sperimentali

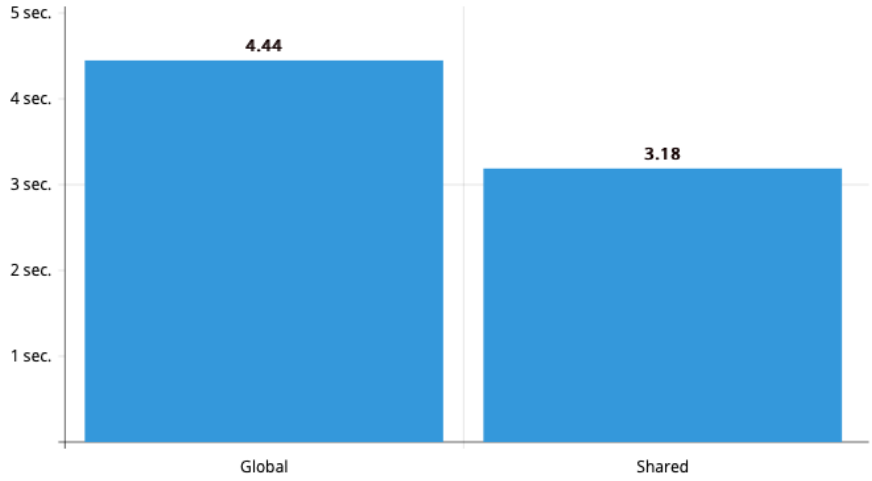
Durante lo sviluppo del progetto ho confrontato i tempi di diverse soluzioni che andiamo a mostrare:

3.1 CPU - GPU:



3.1 Media di 15 lanci

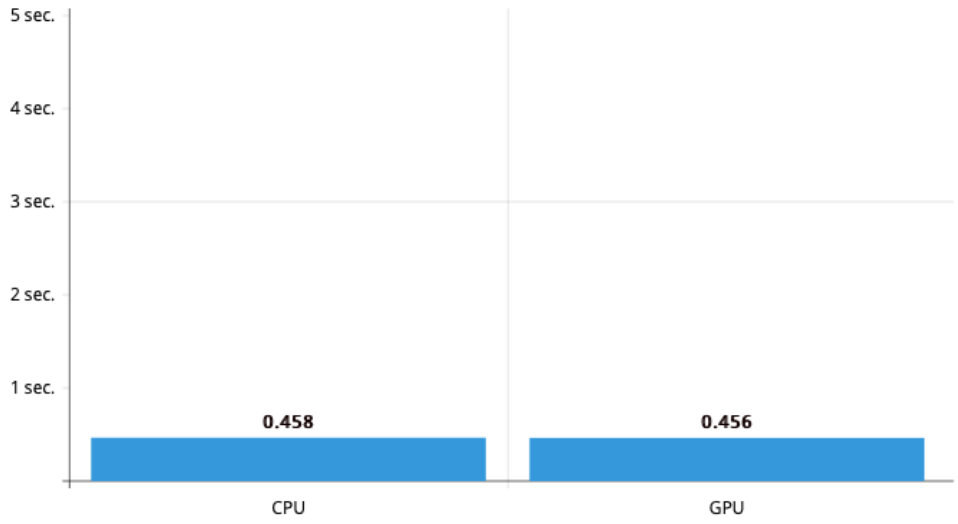
3.2 Template in Shared memory - Template in Global memory:



3.2 Media di 15 lanci ripetuti 30 volte ciascuno
(per evidenziare la prestazione)

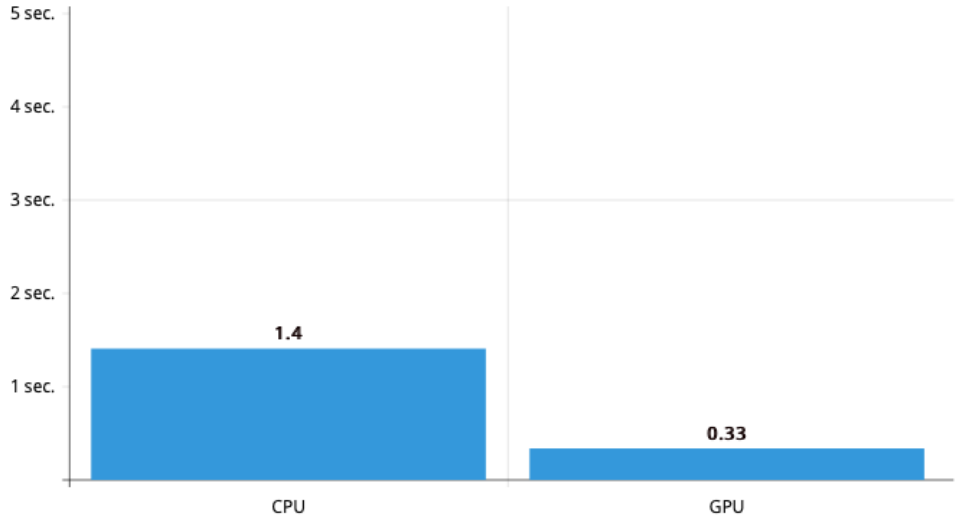
3.3 Ricerca minimo CPU - Ricerca minimo GPU

Come evidenziato anche in precedenza la differenza di prestazioni tra ricerca con CPU e ricerca con GPU è praticamente inesistente per le dimensioni dell'array utilizzato nel progetto.



3.3 Media di 15 lanci

Per dimostrare l'effettiva prestazione del codice parallelo nella ricerca ho comparato la ricerca tra CPU e GPU su un array (matrice) di 1.000.000.000 dimensioni:



3.4 Media di 15 lanci

4. Valutazione e osservazioni

In conclusione durante lo sviluppo del progetto sono state applicate diverse tecniche per migliorare le performance di ricerca del template all'interno di una immagine. Alcune di queste sono state solo testate e poi non introdotte perchè non portavano un significativo miglioramento delle prestazioni.
Come visto in precedenza il punto di forza è sicuramente l'utilizzo della shared Memory per il template invece un punto debole è l'accesso alla memoria globale con problemi di coalescenza per l'immagine origine.
Un possibile miglioramento da apportare in versioni future potrebbe essere quello di spostare il lavoro dei singoli thread non sulla sommatoria delle differenze tra i pixel ma direttamente sulle singole differenze, quindi i thread non calcoleranno un'intera differenza ma solo le singole differenze tra i pixel. Questo introdurrà l'utilizzo di operazioni atomiche sulla somma delle singole differenze.

4.1 Ambiente di test

Per la creazione e test del progetto è stata utilizzata la macchina di laboratorio dell'università con le seguenti caratteristiche:

```
 ---General Information for device 0---
Name: Quadro P600
Compute capability: 6.1
Clock rate: 1556500
Device copy overlap: Enabled
Kernel execution timeout : Enabled
 ---Memory Information for device 0---
Total global mem: -2147483648
Total constant Mem: 65536
Max mem pitch: 2147483647
Texture Alignment: 512
 ---MP Information for device 0---
Multiprocessor count: 3
Shared mem per mp: 49152
Registers per mp: 65536
Threads in warp: 32
Max threads per block: 1024
Max thread dimensions: (1024, 1024, 64)
Max grid dimensions: (2147483647, 65535, 65535)
```

L’immagini utilizzate per i test sono quelle all’interno della cartella immagini con nome “templateC.bmp” e “origine.bmp”. All’interno della cartella si trova un’altro template con nome “templateB.bmp” per effettuare ulteriori prove.

4.2 Guida all'uso

Il programma per funzionare richiede 3 parametri:

1. Percorso dell’immagine origine in formato .BMP;
2. Percorso dell’immagine template in formato .BMP;
3. Percorso dell’immagine in output;

Restituisce in output, nel percorso specificato al punto 3, l’immagine origine con un rettangolo di colore rosso nella posizione dove è stata trovata la corrispondenza.

Istruzioni per compilare ed eseguire vedere il file: *istruzioni.txt*