2025

TEMPLATE MATCHING CON ALGORITMO SSD

TEMPLATE MATCHING E COLOR FILTERING PER IL RICONOSCIMENTO DI UNA REDIDROP

ANDREA FASANO 0001189592
LORENZO PEDRONI 0001190405

Indice

Sommario

Il progetto si pone l'obiettivo di realizzare un'implementazione dell'algoritmo di template matching basato sulla **SSD** (Sum of Squared Differences). Dopo un'approfondita analisi dell'algoritmo, sono state sviluppate diverse versioni: una prima implementazione sequenziale, seguita da una versione **SIMD** e una con **OpenMP**. Successivamente, prima di procedere con l'implementazione in **CUDA**, si è intrapreso un processo di ottimizzazione algoritmica con l'introduzione di matrici integrali e del fattore di cross-correlazione, con l'obiettivo di ridurre drasticamente il numero di operazioni necessarie. Una volta riformulato l'algoritmo in forma ottimizzata, è stata sviluppata una nuova versione sequenziale per poi migrare verso una versione CUDA, su cui è stato infine condotto uno studio delle prestazioni volto a individuare i colli di bottiglia e ottimizzare il parallelismo.

Studio del problema

Il **template matching** è una tecnica di computer vision che ha lo scopo di individuare un'immagine detta **template** all'interno di un'altra immagine più grande, detta **source**. L'algoritmo SSD si presta a questo tipo di applicazione grazie alla sua semplicità concettuale e alla possibilità di parallelizzazione. L'algoritmo è stato testato in un caso d'uso concreto: il riconoscimento automatico di un prodotto industriale in uscita da un macchinario. È stata inoltre realizzata una **versione su video**, simulando un'applicazione reale del programma.

Storicamente, l'algoritmo SSD ha origine tra gli anni '70 e '80, inizialmente sviluppato in ambiti di visione robotica e successivamente utilizzato per **stereo detection** e **template matching**. Nonostante oggi l'approccio sia stato in gran parte sostituito da tecniche di **deep learning**, l'algoritmo SSD rimane un ottimo punto di partenza per comprendere le basi del calcolo parallelo e della visione artificiale tradizionale.

Funzionamento: algoritmo SSD base

L'algoritmo SSD confronta una finestra della source con il template tramite la seguente formula:

$$SSD(x,y) = \sum_{i=0}^{h-1} \sum_{j=0}^{w-1} [I(x+i,\, y+j) - T(i,\, j)]^2$$

dove:

- I(x+i, y+j) è il valore del pixel nella **source**,
- T(i,j) è il valore del pixel nel **template**,

- *(h,w)* sono altezza e larghezza del template,
- *(x,y)* è la coordinata di partenza della finestra nella source.

Le immagini vengono prima convertite in **scala di grigi** per semplificare i calcoli, riducendo ogni pixel a un valore tra 0 (nero) e 255 (bianco). L'algoritmo fa scorrere il template su ogni posizione valida della source, calcolando un valore SSD per ciascuna.

Esempio semplificato

Source 5×5:

css
Copia codice
[1 2 3 2 1]
[4 5 6 5 4]
[7 8 9 8 7]
[4 3 2 3 4]
[1 0 1 2 3]

Template 2×2:

css
Copia codice
[6 5]
[3 2]

Esempio: calcolo SSD in posizione (1,2). Finestra source:

[6 5]
[9 8]

Confronto:

Posizione	I(i,j)	T(i,j)	$(I-T)^2$
(0,0)	6	6	0
(0,1)	5	5	0

(1,0)	9	3	36
(1,1)	8	2	36

Risultato: SSD(1,2) = 0 + 0 + 36 + 36 = 72

Ripetendo l'operazione su ogni finestra, si ottiene una matrice SSD in cui **il valore minimo indica la posizione più simile al template.**

Algoritmo sequenziale

Per implementare il template matching utilizzando il metodo delle **Somme delle Differenze Quadrate (SSD)** è stata realizzata una funzione in C++. Le immagini della sorgente e del template vengono inizialmente convertite in scala di grigi tramite la libreria OpenCV.

Successivamente, le matrici delle due immagini e una matrice di output per i punteggi vengono passate come argomenti alla funzione **templateMatching**.

La funzione scorre **tutte le possibili posizioni del template** sull'immagine sorgente mediante due cicli for annidati. Per ciascuna posizione, altri due cicli for interni confrontano pixel per pixel i valori del template e della regione corrispondente dell'immagine, calcolando la differenza e sommando il quadrato di ciascuna differenza in una variabile score.

Il punteggio SSD ottenuto per ciascuna posizione viene salvato nella matrice dei punteggi (**matchScores**), in corrispondenza delle coordinate correnti.

La dimensione della matrice dei punteggi è **(H − h + 1) × (W − w + 1)**, dove **H** e **W** sono le dimensioni dell'immagine sorgente, h e w quelle del template. Il valore minimo in questa matrice rappresenta la posizione con il miglior matching, individuato tramite la funzione **cv::minMaxLoc()**.

Infine, viene disegnato un rettangolo verde sull'immagine originale nella posizione trovata, per evidenziare graficamente il risultato del template matching.

Dal punto di vista computazionale:

- Scorrere tutte le possibili posizioni ha complessità **O(H × W)**.
- Il confronto del template in ogni posizione ha complessità **O(h × w)**.
- Quindi, la complessità complessiva dell'algoritmo è **O(H × W × h × w)**.

Algoritmo SIMD

Il problema del calcolo della somma delle differenze al quadrato (SSD) tra un'immagine e un template non impone una sequenzialità: ogni finestra dell'immagine può essere confrontata con il template in modo indipendente. Questo lo rende altamente parallelizzabile.

L'approccio SIMD (*Single Instruction, Multiple Data*) consente di processare simultaneamente più dati utilizzando un'unica istruzione, sfruttando i **registri estesi da 128 bit** delle moderne CPU tramite le **istruzioni SSE (Streaming SIMD Extensions)**.

Nel contesto del *template matching*, questo significa che possiamo processare **più pixel contemporaneamente** per ogni iterazione del ciclo annidato. In particolare:

- I pixel dell'immagine e del template sono rappresentati come **uint8_t**, cioè 8 bit per pixel.
- In ogni iterazione SIMD, vengono caricati **16 pixel** per l'immagine e **16 per il template** in due registri a 128 bit tramite **_mm_loadu_si128**.

Poiché l'operazione di differenza tra pixel può generare **valori negativi**, è necessario:

- Espandere ciascun gruppo di 16 pixel da 8 bit a 16 bit con segno.
- Questo avviene tramite le istruzioni **_mm_unpacklo_epi8** e **_mm_unpackhi_epi8**, che generano due registri da 8 elementi a 16 bit ciascuno per immagine e template.

Una volta ottenuti i registri espansi:

- Si effettua la **sottrazione elemento per elemento** per calcolare le differenze.
- Poi si calcola il quadrato di ciascuna differenza e lo si accumula, utilizzando l'istruzione **_mm_madd_epi16**, che moltiplica e somma coppie adiacenti di valori a 16 bit, producendo risultati a 32 bit.

Poiché il risultato finale della SSD potrebbe superare la capacità di un registro a 32 bit, specialmente per template di grandi dimensioni (es. 500x500 pixel), è necessario espandere questi 32 bit in **64 bit** usando **_mm_cvtepu32_epi64**, ottenendo infine **registri a 2 elementi da 64 bit**, che possono accumulare somme molto più grandi evitando overflow.

I valori risultanti vengono poi sommati ed estratti con **_mm_store_si128** e scritti nella matrice dei punteggi.

Infine, se la larghezza del template non è un multiplo di 16, i **pixel rimanenti** vengono processati scalarmente (uno alla volta) per completare il calcolo.

Algoritmo SIMD e openMP

Un ulteriore livello di parallelizzazione è stato ottenuto sfruttando OpenMP, una libreria che consente di distribuire il carico computazionale su più thread.

Nel contesto del template matching, è possibile parallelizzare il ciclo esterno che scorre l'immagine, assegnando ciascuna **riga della finestra di scorrimento** (ovvero ciascuna riga della matrice dei punteggi **matchScores**) a un thread separato.

Questo è stato implementato utilizzando la direttiva: **#pragma omp parallel for**

sul ciclo for più esterno del template matching. Sono state utilizzate le impostazioni di default di OpenMP, che:

- Distribuiscono automaticamente le iterazioni tra tutti i thread disponibili del sistema;
- Utilizzano una politica di scheduling statica (**schedule(static)**), che assegna blocchi di iterazioni in modo uniforme ai thread all'inizio della computazione.

Questa scelta è particolarmente adatta al nostro caso, poiché:

- Ogni iterazione del ciclo esterno ha un carico computazionale costante;
- Lo scheduling statico garantisce un bilanciamento ottimale senza necessità di ribilanciamento dinamico;
- Lo scheduling dinamico introdurrebbe un overhead aggiuntivo inutile in questo contesto.

Speed up

Dato: Immagine 1326×1025 Template 479×432, CPU Intel Core i7-8565U 1.80GHz

Versione	Tempo (ms)	Speed Up
Sequenziale	924.604 ms	
SIMD_64v	543.347 ms	1,7
SIMD_OpenMP	170.222 ms	5,4

Dato: Immagine 1200×1983 Template 150×150, CPU Intel Core i7-8565U 1.80GHz

Versione	Tempo (ms)	Speed Up
Sequenziale	369.821 ms	
SIMD_64	225.578 ms	1,6
SIMD_32	178.734 ms	2,1
SIMD_64_OpenMP	71.522 ms	5,2
SIMD_32_OpenMP	56.726 ms	6,5

Ottimizzazioni dell'algortimo SSD

Dopo le prime versioni (C++, SIMD e OpenMP), è emerso che i tempi di calcolo erano **eccessivamente elevati**: ad esempio, la versione sequenziale C++ inizialmente richiedeva **circa 16 minuti** per completare l'elaborazione.

Introduzione delle Matrici Integrali

Per ottimizzare il calcolo, si è introdotto il concetto di **matrice integrale**, che consente di calcolare somme di sottomatrici in **tempo costante** tramite soli 4 accessi di memoria.

Data una matrice A, la matrice integrale I è definita come:

$$\sum_{i=0}^{x}\sum_{j=0}^{y} A(i,\, j)$$

Esempio pratico:

Matrice A (2×2):

[1 3]
[5 9]

Matrice integrale I:

[1 4]

[6 15]

Riformulazione della SSD con Cross-Correlazione

Analizzando matematicamente l'algoritmo SSD, si può espandere la formula come:

$$\Sigma I^2(x+i,\, y+j) + \Sigma T^2(i,\, j) - 2 \cdot \Sigma[I(x+i,\, y+j) \cdot T(i,\, j)]$$

Da cui derivano tre componenti:

1. **Somma dei quadrati della source nella finestra:** ottenibile in tempo costante usando la matrice integrale al quadrato.
2. **Somma dei quadrati del template:** è **una costante**, calcolata una sola volta.
3. **Cross-correlazione tra source e template:** calcolabile in modo efficiente come prodotto scalare.

Risultati dell'Ottimizzazione

Con l'introduzione delle matrici integrali e della cross-correlazione:

- si è **passati da h· w** sottrazioni e quadrati a **4 accessi + 1 prodotto scalare** per finestra.
- la complessità è scesa da $O(H \cdot W \cdot h \cdot w)$ a $O(2 \cdot H \cdot W + \log_2(H \cdot W))$.

Questo ha portato a **risparmi computazionali enormi**, riducendo potenzialmente **decine di miliardi di operazioni**, rendendo l'algoritmo molto più adatto all'elaborazione parallela e aprendo la strada all'implementazione in CUDA.

Nuovo algoritmo sequenziale

Per implementare la nuova versione dell'algoritmo di template matching è stata sviluppata la libreria tMatchSeq.h, contenente un insieme di funzioni.

La funzione principale, seq_templateMatchingSSD(), riceve in input le matrici dell'immagine e del template (entrambe convertite in scala di grigi) e ne normalizza i valori in virgola mobile (float32) tra 0 e 1, per prevenire fenomeni di overflow durante il calcolo delle immagini integrali.

Successivamente, mediante la funzione computeIntegralImagesSequential(), vengono calcolate le immagini integrali dei quadrati dei pixel dell'immagine e del template. Il calcolo avviene iterando su righe e colonne, sommando i valori in modo cumulativo. Per ogni pixel, il valore viene innanzitutto elevato al quadrato. Successivamente, si procede al calcolo dell'integrale cumulativo: la somma avviene lungo le righe in modo progressivo, e per ciascun pixel delle righe successive alla prima si aggiunge anche il valore corrispondente

della riga superiore (già integrato). In questo modo, ogni valore dell'immagine integrale rappresenta la somma dei quadrati dei pixel dalla posizione (0,0) fino alla posizione corrente.

Dopo aver ottenuto le immagini integrali, viene calcolata la matrice di cross-correlazione tramite la funzione crossCorrelationFFT(). Questa funzione applica la trasformata di Fourier (FFT) sia all'immagine che al template, esegue il prodotto tra l'immagine e il complesso coniugato del template, e infine applica l'antitrasformata (iFFT) per ottenere la mappa di cross-correlazione nello spazio spaziale.

La funzione computeSSDSequentialWithIntegrals() si occupa poi del calcolo della mappa SSD (Sum of Squared Differences), in cui ogni elemento rappresenta il risultato del matching del template in una specifica posizione dell'immagine. Per ogni posizione, il valore SSD viene calcolato secondo la seguente formula:
SSD = S2 - 2 * SC + templateSqSum
dove:

- SC è il valore della convoluzione tra immagine e template ottenuto dalla cross-correlazione;
- templateSqSum è la somma dei quadrati dei pixel del template, ottenuta dall'immagine integrale del template;
- S2 rappresenta la somma dei quadrati dei pixel dell'immagine nella regione coperta dal template, calcolata con la funzione getRegion().

La funzione getRegion() permette di estrarre la somma dei pixel in una sotto-regione dell'immagine utilizzando i valori precalcolati dell'immagine integrale. Per fare ciò, sottrae i contributi delle righe superiori e delle colonne a sinistra della regione d'interesse, e aggiunge il valore dell'angolo in alto a sinistra, precedentemente sottratto due volte durante il calcolo.

Al termine dell'elaborazione, la funzione computeSSDSequentialWithIntegrals() restituisce una mappa SSD completa. Il minimo valore della mappa, indicativo della migliore corrispondenza tra immagine e template, viene individuato tramite la funzione cv::minMaxLoc() di OpenCV, che restituisce anche le coordinate della posizione corrispondente.

Algoritmo Cuda naive

Per implementare la versione parallela del codice sequenziale, sono stati sviluppati diversi kernel CUDA che replicano le funzionalità principali del template matching, sfruttando il

parallelismo massivo offerto dalla GPU. Di seguito viene fornita una descrizione dettagliata dei kernel implementati, in corrispondenza delle funzionalità principali dell'algoritmo.

Calcolo immagini integrali

Per il calcolo delle immagini integrali sono stati realizzati tre kernel:

- *multiply()*: calcola il quadrato dei pixel dell'immagine;
- *rowCumSum()*: calcola la somma cumulativa lungo le righe;
- *colCumSum()*: calcola la somma cumulativa lungo le colonne.

Multiply

Il kernel **multiply** viene lanciato con blocchi 2D di dimensione 32x32 e una griglia tale da coprire l'intera immagine. Ogni thread elabora un singolo pixel: legge il valore del pixel dalla memoria globale, lo eleva al quadrato e scrive il risultato nella stessa posizione nella matrice di output.

rowCumSum e colCumSum

Questi due kernel vengono eseguiti in successione: prima **rowCumSum** calcola le somme cumulative sulle righe, successivamente **colCumSum** le calcola sulle colonne della matrice risultante.
Entrambi i kernel utilizzano blocchi 1D da 32 thread, con la griglia calcolata per coprire l'intero numero di righe o colonne. Ogni thread gestisce una riga (per **rowCumSum**) o una colonna (per **colCumSum**), iterando con un ciclo for sugli elementi e accumulando la somma in una variabile temporanea, ad ogni iterazione del ciclo for il thread scrive il valore nella matrice di output al corrispettivo indice.

Calcolo fattore di cross correlazione

Per il calcolo della cross-correlazione nel dominio delle frequenze è stata utilizzata la libreria cuFFT, integrata con tre kernel CUDA dedicati:

- padToZero(): esegue il padding dell'immagine per adattarla alle dimensioni ottimali della FFT e per prevenire aliasing da convoluzione circolare;
- mulConj(): effettua la moltiplicazione punto a punto tra l'FFT dell'immagine e il complesso coniugato dell'FFT del template;
- normalize(): normalizza il risultato della trasformata inversa FFT.

padToZero

l kernel **padToZero** ha il compito di creare una matrice più grande rispetto all'immagine originale, inizializzandola a zero. Le dimensioni minime della nuova matrice corrispondono a *(dimensione immagine + dimensione template - 1)*, in modo da evitare effetti di aliasing dovuti alla convoluzione circolare. Tuttavia, queste dimensioni vengono successivamente estese fino a raggiungere le **dimensioni ottimali per l'esecuzione della FFT**, per massimizzare l'efficienza computazionale.

Il kernel viene lanciato con blocchi 2D di dimensione 16x16 e una griglia calcolata per coprire l'intera matrice padded. Ogni thread gestisce un singolo elemento della matrice: se si trova all'interno dell'intervallo coperto dall'immagine originale, copia il corrispondente valore; altrimenti, scrive uno zero.

Dopo il padding, si procede con la creazione dei piani FFT tramite **cufftPlan2d** e l'esecuzione della trasformata diretta **cufftExecR2C** sia per l'immagine che per il template.

mulConj

Il kernel **mulConj** calcola la moltiplicazione complessa tra l'FFT dell'immagine e il complesso coniugato dell'FFT del template, secondo la formula:

$$(a+bi)(c-di)=(ac+bd)+(bc-ad)i$$

Il kernel viene lanciato con **blocchi 2D di dimensione 16×16** e una **griglia dimensionata** per coprire l'intera matrice di frequenze. Ogni thread elabora un singolo elemento, calcolando prima il complesso coniugato del valore corrispondente del template, poi effettuando la moltiplicazione complessa con il valore dell'immagine. Il risultato viene salvato in una nuova matrice complessa.

Successivamente, la matrice risultante viene riportata nel dominio spaziale mediante **cufftExecC2R**.

normalize

Poiché cufftExecC2R non esegue la normalizzazione automatica della trasformata inversa, il kernel normalize è responsabile del bilanciamento energetico. Ogni valore risultante viene diviso per il prodotto delle dimensioni della matrice (m * n), dove m e n rappresentano le dimensioni padded iniziali. Il kernel viene lanciato con blocchi 1D da 256 thread, ciascuno dei quali elabora un singolo elemento della matrice.

Calcolo della matrice SSD dei risultati

Il kernel **computeSSD** si occupa della creazione della matrice dei risultati contenente i valori SSD (Sum of Squared Differences) tra l'immagine e il template in tutte le posizioni possibili.

La logica è analoga a quella dell'implementazione sequenziale: ogni thread CUDA calcola il valore SSD relativo a una singola posizione dell'immagine, corrispondente a un possibile allineamento del template.

Il kernel utilizza una funzione device **getRegionSum** per calcolare la somma dei quadrati dei pixel dell'immagine nella sottofinestra corrente, sfruttando la **matrice integrale imageSqSum**. Questo valore è indicato come **S2**.

Il termine **SC** rappresenta il valore della **cross-correlazione** tra immagine e template nella stessa posizione, ed è pre-calcolato e salvato nella matrice **crossCorrelation**. Questa matrice è **paddata**, pertanto l'accesso viene corretto usando **paddedCols** come stride nel calcolo dell'indice.

Il valore **templateSqSum** è un valore costante, rappresenta la somma dei quadrati dei valori del template. La formula finale applicata è:

SSD = S2 - 2 * SC + templateSqSum

Il kernel viene lanciato con **blocchi 2D da 32×32 thread**, e la griglia è dimensionata per coprire la **matrice dei risultati** di dimensione **(width - kx + 1) × (height - ky + 1)**, ovvero tutte le possibili posizioni in cui il template può essere contenuto nell'immagine.

Color filtering

La funzione di *color filtering* è stata introdotta per estendere il progetto e, soprattutto, per affrontare una dinamica concreta: volevamo che il nostro programma fosse in grado di riconoscere se all'interno delle capsule rilevate fosse presente un prodotto colorato (in particolare, sapone per il corpo di colore blu). Questo ci consente di determinare se le capsule siano piene o vuote.

Funzionamento del color filtering

Questa funzione viene eseguita successivamente al *template matching*, analizzando la porzione dell'immagine sorgente individuata come la migliore corrispondenza (cioè quella con il valore SSD minimo). L'algoritmo è identico sia nella versione sequenziale che in quella CUDA, e funziona come segue.

È stata definita inizialmente una funzione *wrapper* per:

- inizializzare le variabili,
- allocare la memoria,
- (nella versione CUDA) lanciare i kernel.

Di seguito spieghiamo i due kernel principali, evidenziando le differenze tra versione sequenziale e parallela.

convertBGRtoUchar3Kernel

Questo kernel converte i pixel della *ROI* (Region Of Interest) da formato BGR a uchar3, un formato più efficiente in CUDA poiché consente di gestire separatamente i tre canali di colore (Blue, Green, Red). Questo è essenziale per i passaggi successivi.

- **Configurazione**: blockSize(16, 16) e gridSize((image.cols + blockSize.x - 1) / blockSize.x , (image.rows + blockSize.y - 1) / blockSize.y).
  Questo assicura che ogni pixel venga gestito da un thread dedicato, minimizzando il numero di thread inattivi.
- **Calcolo dell'indice del thread**: viene fatto tramite coordinate globali, più intuitive nel caso di dati bidimensionali.

Nella versione sequenziale, questa fase è gestita direttamente durante il confronto, utilizzando il tipo Vec3b di OpenCV che consente accesso ai singoli canali RGB.

countBluePixelKernel

Questo kernel effettua il conteggio dei pixel blu:

- Converte ogni pixel da BGR a HSV (Hue, Saturation, Value), un modello di colore più stabile in presenza di variazioni di luce.
- Confronta i valori HSV con soglie predefinite.
- Utilizza atomicAdd per incrementare un contatore condiviso.
- **Configurazione**: dimBlock(16,16) e gridSize((roi.width + blockSize.x - 1) / blockSize.x , (roi.height + blockSize.y - 1) / blockSize.y).
  Questo limita i calcoli e confronti alla sola ROI, migliorando l'efficienza.

Ottimizzazioni

Un'analisi approfondita tramite **NVIDIA Nsight Compute** ha evidenziato un collo di bottiglia nella prima versione CUDA, localizzato nei kernel per il calcolo delle matrici integrali: rowCumSum e colCumSum.

Problemi riscontrati:

- Accessi sequenziali e non coalescenti alla memoria globale, causando stalli.
- Utilizzo inefficiente dei thread (es. un solo thread per riga).

- Kernel *memory bound*, confermato dall'analisi del *Floating Point Operations Roofline*.

Soluzione: algoritmo Prefix Sum ottimizzato

Abbiamo adottato un nuovo approccio per il calcolo delle immagini integrali, basato sul **prefix sum** 2D:

1. Calcolo delle somme cumulative per riga.
2. Trasposizione della matrice.
3. Calcolo delle somme cumulative per colonna (sull'immagine trasposta).
4. Nuova trasposizione per tornare alla forma originale.

Questa strategia ha:

- Ridotto gli accessi non coalescenti.
- Ottimizzato l'utilizzo della memoria condivisa e dei registri.

Nuove funzioni CUDA

__device__ __forceinline__ float warpPrefixSum(float val)

Questa funzione esegue un prefix sum all'interno di un **warp** (gruppo di 32 thread). Ogni thread accumula i valori dei thread precedenti tramite un ciclo for, in cui l'**offset** raddoppia a ogni iterazione (1, 2, 4, 8, …). L'ottimizzazione avviene tramite:

- **unroll automatico** del ciclo, per favorire l'esecuzione in parallelo;
- uso di __shfl_up_sync, che consente a ogni thread di accedere direttamente al dato di un altro thread dello stesso warp, **senza usare la memoria condivisa**;
- uso della direttiva __forceinline__, che forza l'inlining da parte del compilatore, riducendo l'overhead di chiamata.

__device__ float blockPrefixSum(float val, float* sharedMem)

Questa funzione gestisce il prefix sum **intra-blocco**. La logica è strutturata in tre fasi:

1. Ogni thread esegue il warpPrefixSum all'interno del proprio warp.
2. L'ultimo thread di ciascun warp scrive la somma cumulativa in **memoria condivisa**.
3. Si esegue un prefix sum sui valori scritti dai warp, per ottenere gli **offset globali**.
4. Ogni thread aggiunge l'offset del proprio warp al valore calcolato nel primo step.

Questo approccio permette una **scalabilità efficiente** all'interno di blocchi di dimensioni elevate, riducendo la necessità di sincronizzazioni costose.

__global__ void optimizedRowPrefixSum(float* input, float* output, float* blockSums, int width, int height)

Questo kernel calcola il prefix sum **per riga** dell'immagine:

- Ogni blocco gestisce **una singola riga**.
- I pixel della riga vengono caricati in **shared memory**, elaborati con blockPrefixSum, e salvati in output.
- La **somma finale della riga** viene memorizzata nell'array blockSums, che servirà per calcolare gli offset tra blocchi (passaggio successivo).

__global__ void addBlockOffsets(float* data, float* blockSums, int width, int height)

Questo kernel **aggiunge gli offset tra i blocchi**, completando il prefix sum lungo le righe:

- Viene saltato il **blocco 0** (che non necessita offset).
- Per ciascun blocco viene recuperato il relativo offset da blockSums e **aggiunto a tutti i suoi elementi**.

A questo punto, il prefix sum è completo lungo l'asse delle righe. Si passa quindi alle colonne.

__global__ void optimizedTranspose(float* input, float* output, int width, int height)

Questo kernel effettua la **trasposizione della matrice**, necessaria per riutilizzare il calcolo delle righe anche per le colonne (in un secondo passaggio).

- I dati vengono gestiti in **tile da 32×33**: la dimensione 33 (anziché 32) sulla seconda dimensione **evita i bank conflicts** durante l'accesso alla memoria condivisa.
- Dopo aver caricato i tile in shared memory, viene eseguita la trasposizione vera e propria.
- Infine, i dati trasposti vengono copiati nella memoria globale.

Questo passaggio viene effettuato due volte:

1. Dopo il prefix sum per riga, per trasporre l'immagine.
2. Dopo il prefix sum per colonna, per riportare l'immagine al formato originale.

cudaError_t computeIntegralImageOptimized(float* d_input, float* d_output, int width, int height)

Questa funzione *wrapper* lancia i kernel in sequenza con i parametri corretti:

1. Calcolo delle dimensioni ottimali in base alla dimensione dell'immagine.
    a. **Thread per blocco**: 256 (multiplo di 32).
    b. **Grid size**: calcolata per fare in modo che ogni blocco gestisca una riga.
2. Lancio di:
    a. optimizedRowPrefixSum
    b. addBlockOffsets
3. Esecuzione del primo optimizedTranspose
4. Ripetizione del calcolo (step 1 e 2) sulla matrice trasposta
5. Esecuzione del secondo optimizedTranspose per riportare il risultato alla forma iniziale

Questo schema garantisce un **calcolo completo ed efficiente** dell'immagine integrale, ottimizzando:

- accessi alla memoria (grazie al memory coalescing e all'uso della shared memory),
- uso delle risorse hardware (warp-level primitives e sincronizzazione ridotta),
- riuso del codice (eseguendo lo stesso prefix sum per righe anche sulle colonne trasposte).

Ottimizzazione della conversione da BGR a Uchar3

La versione iniziale della funzione di conversione presentava limiti:

- Ogni thread gestiva un solo pixel.
- Accessi in memoria potenzialmente non coalescenti.
- Dipendenza dalla definizione precisa della griglia.

Nuova versione ottimizzata:

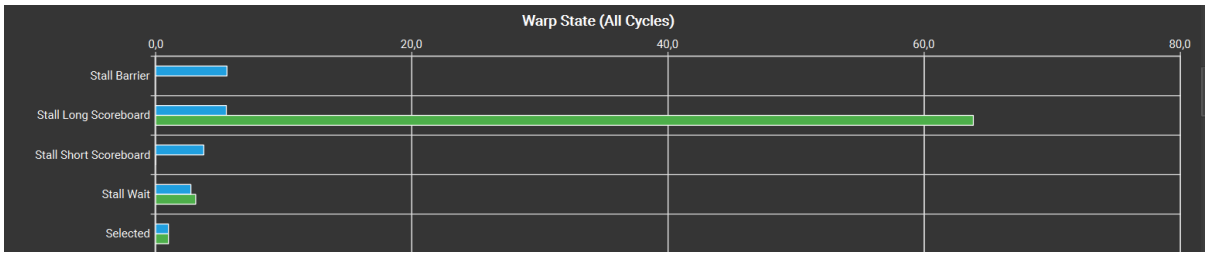
- Ogni thread può gestire più pixel con un loop:
  for (int pixel_idx = tid; pixel_idx < total_pixels; pixel_idx += stride)
- Thread consecutivi accedono a pixel consecutivi, migliorando il *memory coalescing*.
- Utilizza una griglia 1D di dimensione fissa, adatta a immagini di qualsiasi dimensione.
- Massimizza l'occupancy e sfrutta meglio cache L1/L2.

Conclusioni sulle ottimizzazioni

Queste ottimizzazioni hanno migliorato significativamente le prestazioni del sistema, in particolare nel calcolo delle immagini integrali e nel filtering dei colori. L'approccio CUDA ottimizzato, unito alle scelte mirate nella gestione della memoria e della parallelizzazione, ha permesso di superare i colli di bottiglia iniziali e ottenere un'elaborazione più veloce ed efficiente.

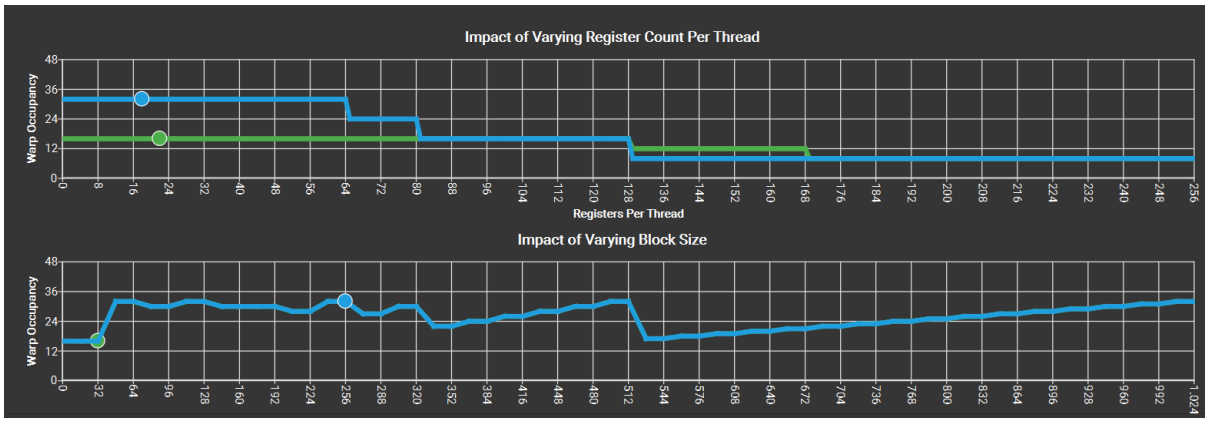
Le seguenti foto sono prese dal confronto dei due report di Nsight Compute e vogliono mostrare i miglioramenti apportati in termini di: warp stall, occupancy e throughtput.

Miglioramento sugli stalli



Come si può notare dal grafico abbiamo ottenuto un grande miglioramento sugli stalli dovuti all'attesa di una dipendenza sulle memorie: globale, condivisa, cache L1 e texture.

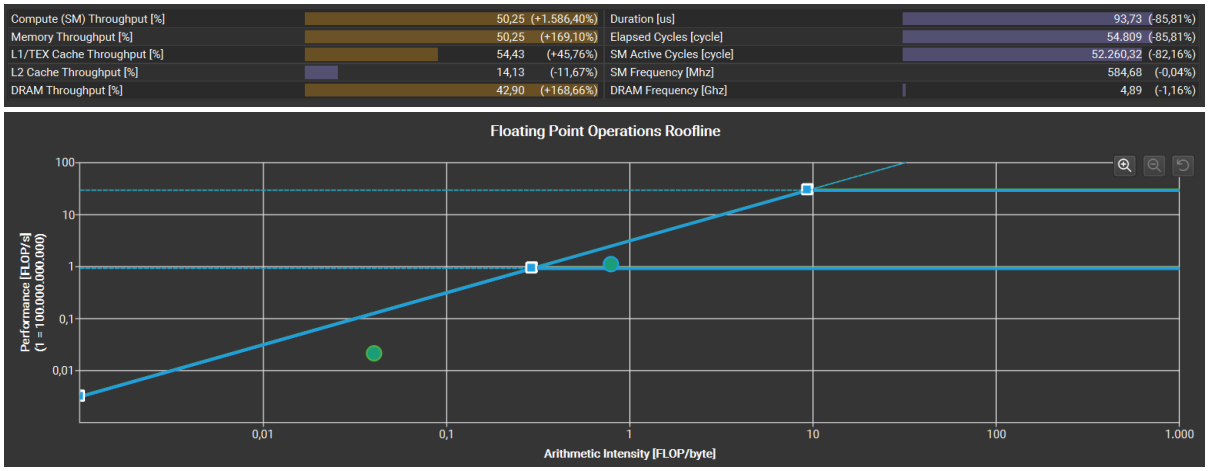
Miglioramento dell'occupancy



Theoretical Occupancy [%]	100	(+100,00%)
Theoretical Active Warps per SM [warp]	32	(+100,00%)
Achieved Occupancy [%]	85,55	(+2.627,23%)
Achieved Active Warps Per SM [warp]	27,38	(+2.627,23%)

Si nota un grande guadagno in termini di occupancy, dovuto anche ad una nuova scelta dei parametri di lancio dei kernel.

Miglioramento del Throughtput



Notiamo un notevole miglioramento anche in termini di throughtput in memoria grazie all'utilizzo dei registri interni ai thread e della memoria condivisa; mentre, il throughtput computazionale aumenta grazie ad una migliore distribuzione dei thread sull'SM garantendo una migliore distribuzione del carico. Infine, notiamo dal Floating Point Operation Roofline che abbiamo un kernel ancora memory bound ma con un notevole aumento dell'AI che passa da un valore di 0,04 a 0,80.

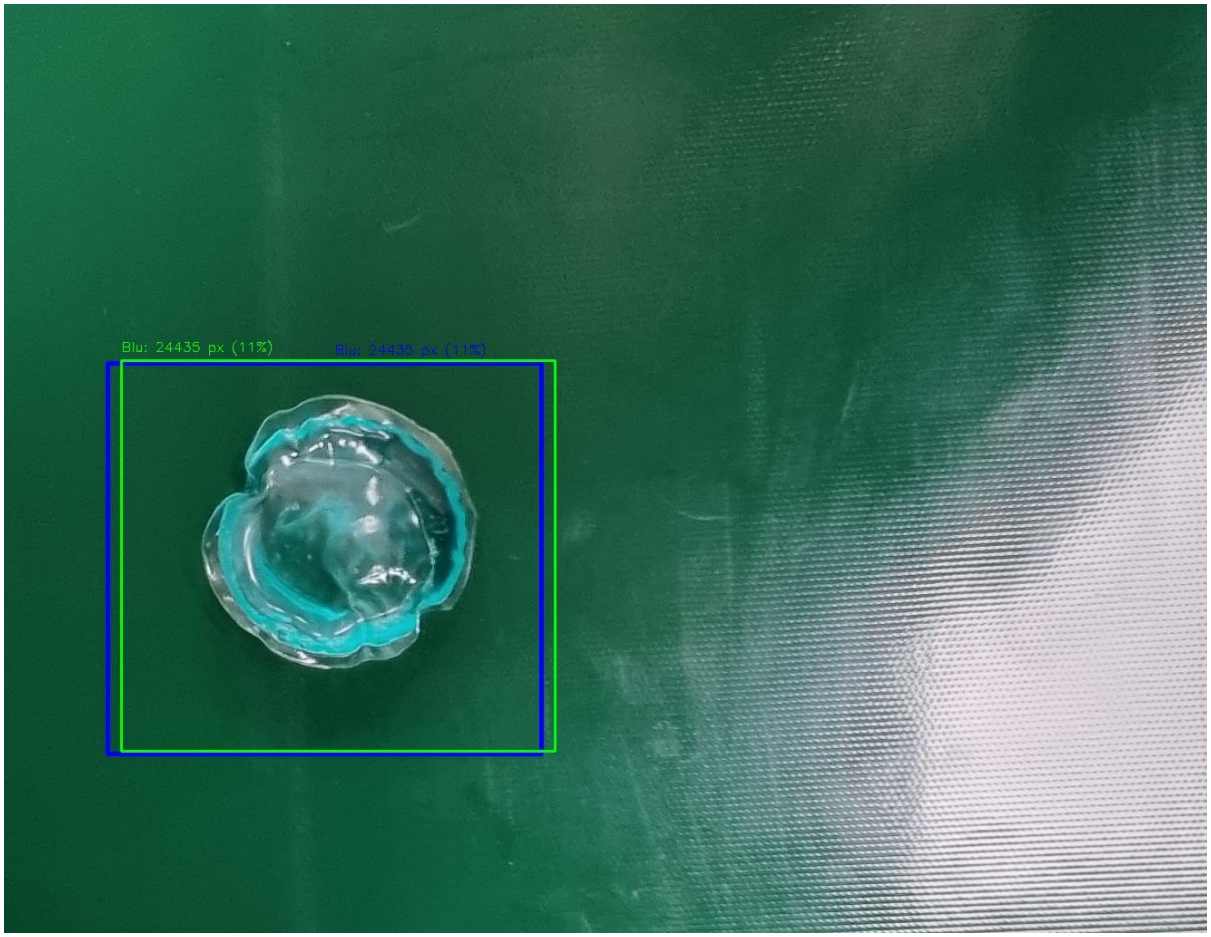
Speedup finali

La seguente tabella mostra gli speedup delle versioni: CUDA naive e CUDA ottimizzata. Calcolati come speedup rispetto alla versione sequenziale ottimizzata.

Speedup tra le varie versioni					
Sezione algoritmo	Sequenziale - tempo in us	CUDA naive - tempo in us	CUDA ottimizzata - tempo in us	Speedup seq/naive	Speedup seq/ottimizzata
Caclolo immagine source integrale	18000	1305,06	565,76	13,8	31,8
Calcolo immagine template integrale	2000	568,35	98,24	3,5	20,4
Cacolo coeff di crosscorrelazione	321000	2514,53	2506,3	127,7	128,1
Computazione per SSD	17000	50,24	51,71	338,4	328,8
Conteggio dei pixel blu	11432	115,61	91,65	98,9	124,7

Risultati

In questa sezione mostriamo la foto risultato dopo l'esecuzione del programma, il riquadro verde rappresenta la ricerca fatta con l'algoritmo CUDA ottimizzato, quello blu con l'algoritmo sequenziale ottimizzato.



Template Matching su Video

Terminata la fase di ottimizzazione, abbiamo deciso di testare il nostro programma su un video. Considerata la natura concreta del caso di studio affrontato, questa estensione rappresenta un'evoluzione naturale del progetto.

L'algoritmo utilizzato è lo stesso descritto nelle sezioni precedenti di questa relazione, incluse le ottimizzazioni introdotte nel capitolo precedente. Di seguito analizziamo le modifiche apportate per gestire l'analisi frame per frame del video. In particolare, abbiamo realizzato:

- Una struttura Tracker che elabora la posizione trovata dal Template Matching e in particolare:
    - Filtra il risultato scartando i valori con SSD < 0,5 e con un numero di pixel blu rilevati minore di 1000
    - Aggiorna la posizione del rettangolo tramite il metodo updatePosition() smorzando la velocità di movimento per migliorare l'output agli occhi dell'utente

- La funzione processVideo() che si occupa di elaborare il video frame per frame utilizzando la classe VideoCapture di OpenCV che dispone di metodi come open e read per l'analisi video dei frame del video.

Conclusioni

In conclusione, possiamo dire di avere una chiara visione di come le tecnologie utilizzate e gli algoritmi adottati influenzino la velocità computazionale nonostante tutte le soluzioni arrivino ad un risultato finale corretto e consistente. Abbiamo ottenuto notevoli speedup tra le varie versioni tramite un'accurata analisi dei colli di bottiglia e, per la parte CUDA, abbiamo cercato di risolvere i problemi maggiori ottenendo buoni risultati, riuscendo a sfruttare in maniera ottimale i vantaggi della parallelizzazione.