



UNIVERSITÀ DEGLI STUDI DI FIRENZE
SCUOLA DI INGEGNERIA - DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Informatica
Parallel Computing - Professor Marco Bertini

KERNEL IMAGE PROCESSING CON JAVA E CUDA

Lorenzo Meoni
Alessio Corsinovi

Anno Accademico 2022/2023

Contents

Introduzione	i
1 Kernel image processing	1
2 Algoritmi implementati e test	3
2.1 Algoritmi implementati	3
2.1.1 Versione sequenziale in Java	3
2.1.2 Versione parallela in Java	5
2.1.3 Versione parallela in CUDA	6
2.2 Risultati dei test	11
3 Conclusioni	16
Bibliografia	17

Introduzione

Lo scopo del progetto è quello di sviluppare un'applicazione che possa essere utilizzata per eseguire kernel image processing. L'applicazione è stata sviluppata in una versione sequenziale (in Java) e in 2 versioni parallele (una in Java e una in CUDA). Sono stati applicati vari tipi di kernel, con dimensioni anche diverse: da Edge-Detect a Blur. Ed è stato valutato lo speedup tra le varie versioni parallele e sequenziale al variare del numero dei thread e delle metodologie utilizzate.

Chapter 1

Kernel image processing

Nell'elaborazione digitale delle immagini, una matrice di convoluzione, kernel, o maschera è una piccola matrice usata per applicare filtri ad immagini. Risulta dunque utile per la sfocatura, affilatura, goffratura, riconoscimento dei contorni e altro ancora. Attraverso l'applicazione della convoluzione di due matrici bidimensionali di cui la prima rappresenta l'immagine originale e la seconda, detta kernel, rappresenta il filtro da applicare. Le matrici kernel sono soprattutto di dimensione dispari, in quanto nella convoluzione è importante identificare il centro della matrice kernel, cosa che avviene facilmente con dimensioni dispari; per esempio, possono essere di dimensione 3x3, 5x5, 7x7, e così via. Difficilmente le matrici kernel sono di grandi dimensioni. [1]

In generale, la convoluzione è un'operazione matematica (*) che, date due funzioni, ne produce una terza che è calcolata come segue:

$$f(t) * g(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau$$

Nelle applicazioni grafiche, si usano convoluzioni su funzioni bidimensionali dove ogni funzione corrisponde a un canale e dove ogni funzione è rappresentabile come funzione a valori discreti (per esempio, 0-255) su domini limitati (pari alla risoluzione delle immagini). L'equazione precedente diventa quindi:

$$w * f(x, y) = \sum_{s=-a}^a \sum_{t=-b}^b w(s, t)f(x - s, y - t)$$

Dove w è la maschera di convoluzione e $f(x, y)$ rappresenta l'immagine. Per ogni pixel dell'immagine, la maschera viene centrata su quel pixel, e vengono moltiplicati i valori nella matrice con i valori dei pixel corrispondenti, e tutti i risultati vengono sommati per ottenere il nuovo valore del pixel corrente della nuova immagine. 1.1

I(0,0)	I(0,1)	I(0,2)	I(0,3)	I(0,4)	I(0,5)	I(0,6)
I(1,0)	I(1,1)	I(1,2)	I(1,3)	I(1,4)	I(1,5)	I(1,6)
I(2,0)	I(2,1)	I(2,2)	I(2,3)	I(2,4)	I(2,5)	I(2,6)
I(3,0)	I(3,1)	I(3,2)	I(3,3)	I(3,4)	I(3,5)	I(3,6)
I(4,0)	I(4,1)	I(4,2)	I(4,3)	I(4,4)	I(4,5)	I(4,6)
I(5,0)	I(5,1)	I(5,2)	I(5,3)	I(5,4)	I(5,5)	I(5,6)
I(6,0)	I(6,1)	I(6,2)	I(6,3)	I(6,4)	I(6,5)	I(6,6)

x

w(0,0)	w(0,1)	w(0,2)
w(1,0)	w(1,1)	w(1,2)
w(2,0)	w(2,1)	w(2,2)

=

G(0,0)	G(0,1)	G(0,2)	G(0,3)	G(0,4)
G(1,0)	G(1,1)	G(1,2)	G(1,3)	G(1,4)
G(2,0)	G(2,1)	G(2,2)	G(2,3)	G(2,4)
G(3,0)	G(3,1)	G(3,2)	G(3,3)	G(3,4)
G(4,0)	G(4,1)	G(4,2)	G(4,3)	G(4,4)

Kernel

Source Image **Output Image**

Figure 1.1: Esempio applicazione kernel 3x3 su un pixel dell'immagine

Chapter 2

Algoritmi implementati e test

2.1 Algoritmi implementati

L'applicazione è stata sviluppata in una versione sequenziale (in Java) e in 2 versioni parallele (una in Java e una in CUDA). I test, per la versione sequenziale e parallela con Java, sono stati eseguiti variando: la dimensione dell'immagine (480p, FullHD, 4K), la dimensione del kernel (3x3, 5x5, 7x7), il numero dei thread (da 2 a 8) e il modo con il quale dividere i pixel da affidare ad i vari thread (abbiamo diviso l'immagine per righe e per colonne). Mentre, per la versione con CUDA, i test sono stati eseguiti variando la dimensione del kernel (3x3, 5x5, 7x7) e realizzando 2 versioni dell'algoritmo che sfruttano o meno la shared e la costant memory.

2.1.1 Versione sequenziale in Java

Nella versione sequenziale del programma sono stati applicati kernel di varie dimensioni (3x3, 5x5 e 7x7) ad immagini di diverse dimensioni (480p, FullHD, 4K) misurando i tempi di esecuzione di ciascuna variante.

Di seguito una parte di codice che mostra l'applicazione del kernel ad un

pixel dell'immagine.

```
// applico il kernel kernelLen x kernelLen sul pixel dell'immagine
for (int m = 0; m < kernelLen; m++) {
    for (int n = 0; n < kernelLen; n++) {
        // calcolo delle coordinate per i pixel al bordo
        int aCoordinate = (a - kernelLen / 2 + m + wid) % wid;
        int bCoordinate = (b - kernelLen / 2 + n + hgt) % hgt;

        int rgbTotal = inputImg.getRGB(aCoordinate, bCoordinate);

        int rgbRed = (rgbTotal >> 16) & 0xff;
        int rgbGreen = (rgbTotal >> 8) & 0xff;
        int rgbBlue = (rgbTotal) & 0xff;

        redFloat += (rgbRed * kernel.kernelMatrix[m][n]);
        greenFloat += (rgbGreen * kernel.kernelMatrix[m][n]);
        blueFloat += (rgbBlue * kernel.kernelMatrix[m][n]);
    }
}
// calcolo sul pixel dell'immagine
Color color = new Color(redOutput, greenOutput, blueOutput);
outputImg.setRGB(a, b, color.getRGB());
```

L'operazione viene ripetuta per tutti i pixel dell'immagine ottenendo l'immagine filtrata. In figura è riportata l'immagine originale 2.1 e nelle figure 2.1 e 2.2 i risultati di alcuni filtri che le sono stati applicati.



Figure 2.1: a sinistra: Immagine originale, qualità 4k, su cui sono stati eseguiti tutti i test di kernel image, a destra: immagine dopo l'applicazione di un filtro di edge detection



Figure 2.2: a sinistra: Immagine, qualità 4k, su cui è stato applicato un filtro di Blur, a destra: immagine su cui è stato applicato un filtro di Sharpen

2.1.2 Versione parallela in Java

Le prestazioni dell'applicazione possono essere migliorate utilizzando un approccio multithread. L'algoritmo di convoluzione dipende dai valori del pixel dell'immagine sorgente e dai valori della matrice di kernel. Questi valori vengono solo letti e non modificati. Inoltre, i valori dei pixel di output vengono scritti nella matrice di output solo una volta e soltanto da un thread. Pertanto, questa operazione è un problema imbarazzantemente parallelo: ogni thread può leggere, eseguire l'operazione e scrivere l'output in memoria senza necessità di sincronizzazione con gli altri thread. Utilizzando questo approccio, con la fase di fork vengono lanciati i thread e con il join aspettiamo il risultato. Ogni thread esegue l'operazione di convoluzione su una sequenza di pixel dell'immagine sorgente. I pixel vengono suddivisi tra i vari thread in modo che ciascuno ottenga la stessa quantità di lavoro da eseguire.

Nella versione parallela del programma sono stati applicati gli stessi kernel e testate le stesse immagini della versione sequenziale, ovvero varie dimensioni (3x3, 5x5 e 7x7) ad immagini di diverse dimensioni (480p, FullHD, 4K). Per ogni combinazione di queste caratteristiche sono stati misurati i tempi di esecuzione. Sono stati misurati inoltre i tempi di esecuzione al variare del numero di thread e al variare del modo in cui i pixel dell'immagine venivano affidati ai vari thread, dividendo l'immagine in righe o in colonne.

Di seguito una parte di codice che mostra la suddivisione dell'immagine in righe, per affidare ad ogni thread, parte dei pixel dell'immagine stessa sui quali operare la convoluzione:

```
int threads = Runtime.getRuntime().availableProcessors();
for (int t=1 ; t<= threads; t++) {
    long start = System.currentTimeMillis();
    MyThread[] thread = new MyThread[t];
    // divido immagine per righe
    int chunkLength = hgt / t;
```

```

int [] bags = new int[t];
// distribuisco gli elementi dividendo l'immagine per righe
Arrays.fill(bags, chunkLength);
for (int rest = hgt % t; rest > 0; rest--) {
    bags[bags.length - rest] += 1;
}

int [] rangeValues = new int[t];
for (int i = 0; i < bags.length; i++) {
    for (int j = 0; j <= i; j++) {
        rangeValues[i] += bags[j];
    }
}

// Lancia i threads - ognuno con il proprio chunks di dati:
thread[0] = new MyThread(0, rangeValues[0] - 1);
thread[0].start();
for (int i = 0; i < t - 1; i++) {
    thread[i + 1] = new MyThread(rangeValues[i], rangeValues[i + 1] - 1);
    thread[i + 1].start();
}

// Aspetto la fine dei thread:
for (int i = 0; i < t; i++) {
    thread[i].join();
}
}

```

La suddivisione viene eseguita utilizzando un numero di thread da 1 a quelli disponibili nel processore dove viene lanciato il programma, nel nostro caso 8.

2.1.3 Versione parallela in CUDA

Il passo successivo nel nostro lavoro è consistito nell'implementare un programma traendo vantaggio dalla tecnologia CUDA e GPU NVidia. [2] Grazie a questa architettura siamo riusciti ad ottenere notevoli miglioramenti rispetto al programma sequenziale ed alla versione parallela con CPU. Il parallelismo CUDA necessita di due “attori”: la parte host che gira su CPU, e il device, dove viene eseguito il calcolo.

L'approccio generale da seguire in questi casi consiste nell'inviare l'immagine ed il kernel al device (cioè alla GPU), eseguire i calcoli sul device (chiamando

un metodo chiamato kernel) e restituire i risultati all'host.

Il kernel contiene fondamentalmente il calcolo della convoluzione. Quando viene chiamato dall'host viene specificata la dimensione della griglia e dei blocchi contenenti i thread da eseguire contemporaneamente. Quello che succede è che ogni thread esegue il codice del kernel allo stesso modo e ognuno di questi thread è caratterizzato da un indice che lo identifica.

Abbiamo utilizzato un primo kernel per scrivere una versione parallela, chiamato filterImageGlobal dove la colonna e la riga vengono identificati dal thread e dal blocco in esecuzione:

- row = blockIdx.y * blockDim.y + threadIdx.y;
- col = blockIdx.x * blockDim.x + threadIdx.x;

Sotto viene riportato il codice del kernel.

```
--global__ void filterImageGlobal(float* d_sourceImagePtr, float* d_maskPtr, float* d_outImagePtr,
    int width, int height, int paddedWidth, int paddedHeight, int filterWidth, int filterHeight){

    const int s = floor(static_cast<float>(filterWidth) / 2);
    const int i = blockIdx.y * blockDim.y + threadIdx.y + s;
    const int j = blockIdx.x * blockDim.x + threadIdx.x + s;

    unsigned int filterRowIndex = 0;
    unsigned int sourceImgRowIndex = 0;
    unsigned int sourceImgIndex = 0;
    unsigned int maskIndex = 0;
    float pixelSum = 0;
    // Check out of bounds thread idx
    if (j >= s && j < paddedWidth - s && i >= s && i < paddedHeight - s) {
        int outPixelPos = (j - s) + (i - s) * width;
        // Apply convolution
        for (int h = -s; h <= s; h++) {
            filterRowIndex = (h + s) * filterWidth;
            sourceImgRowIndex = (h + i) * paddedWidth;
            for (int w = -s; w <= s; w++) {
                sourceImgIndex = w + j + sourceImgRowIndex;
                maskIndex = (w + s) + filterRowIndex;
                pixelSum += d_sourceImagePtr[sourceImgIndex] * d_maskPtr[maskIndex];
            }
        }
        // Thresholding overflowing pixel's values
        if (pixelSum < 0) {
            pixelSum = 0;
        }
    }
}
```

```

    }
    else if (pixelSum > 255) {
        pixelSum = 255;
    }
    // Write pixel on the output image
    d_outImagePtr[outPixelPos] = pixelSum;
}
}

```

Abbiamo poi cercato di migliorare questo kernel agendo sulla gestione della memoria. Questo può essere fatto in due modi: utilizzando la costant memory [3] e la shared memory [4]. La costant memory è un tipo di memoria ad alta velocità in cui è possibile archiviare i dati che non verranno modificati nell'ambito dell'esecuzione del kernel che la utilizza. Questa memoria è come una cache che può essere utilizzata per memorizzare alcune variabili costanti e quindi limitare il numero di letture dalla memoria globale da parte del device, aumentando le prestazioni del programma. Nel nostro sviluppo ci abbiamo memorizzato la matrice con il filtro da applicare all'immagine, che in effetti resta costante per ogni thread che applica la convoluzione.

La shared memory è un altro tipo di memoria della GPU NVidia ed anche questa ha una latenza molto bassa perché è un on-chip costruito su ogni blocco. Anche per questo motivo ha una capacità molto limitata e quindi la sua gestione è davvero importante per poterne usufruire.

La strategia di gestione della memoria condivisa è chiamata tiling: si basa sulla suddivisione dei dati in tile (in blocchi), spostando un tile dalla memoria globale alla shared memory del blocco, eseguendo i calcoli e ripetendo per il tile successivo.

Poiché la memoria condivisa può essere indirizzata da tutti i thread dello stesso blocco, riduciamo drasticamente il numero di chiamate alla memoria globale. Inoltre, affinché questo approccio sia efficace, i thread devono essere sempre sincronizzati, e questo significa che abbiamo un altro punto di possibile rallentamento.

In ogni caso, un buon utilizzo della shared memory mostra un notevole incremento rispetto alle prestazioni di un programma sequenziale o parallelo con CPU.

Sostanzialmente nel codice abbiamo definito un kernel chiamato filterImageShared che tenta di sfruttare sia la costant memory che la shared memory. La dimensione dei tile è stata impostata come 32x32.

```
--device-- __constant__ float d_cFilterKernel[MAX_FILTER_SIZE * MAX_FILTER_SIZE];

__global__ void filterImageShared(float* d_sourceImagePtr, float* d_outImagePtr, int paddedWidth, int
    paddedHeight, int blockWidth, int blockHeight, int surroundingPixels, int width, int height,
    int filterWidth, int filterHeight){

    // Ogni blocco condivide gli stessi dati, consentendo un accesso piu' rapido alla memoria
    // L'accesso alla memoria globale per ciascun blocco sara': numero_di_tile x thread
    // invece della dimensione del kernel x thread

    // Tile shared array
    extern __shared__ float s_data[];

    unsigned int tileSize = blockWidth + 2 * surroundingPixels;
    unsigned int tileHeight = blockHeight + 2 * surroundingPixels;

    // numero di sotto blocchi
    unsigned int noSubBlocks = static_cast<int>(ceil(static_cast<float>(tileHeight) / static_cast
        <float>(blockDim.y)));

    // Get start and end coordinates for blocks
    unsigned int blockStartCol = blockIdx.x * blockWidth + surroundingPixels;
    unsigned int blockEndCol = blockStartCol + blockWidth;
    unsigned int blockStartRow = blockIdx.y * blockHeight + surroundingPixels;
    unsigned int blockEndRow = blockStartRow + blockHeight;

    // Get start and end coordinates for tiles
    unsigned int tileStartCol = blockStartCol - surroundingPixels;
    unsigned int tileEndCol = blockEndCol + surroundingPixels;
    unsigned int tileEndClampedCol = min(tileEndCol, paddedWidth);

    unsigned int tileStartRow = blockStartRow - surroundingPixels;
    unsigned int tileEndRow = blockEndRow + surroundingPixels;
    unsigned int tileEndClampedRow = min(tileEndRow, paddedHeight);

    // Pixel position in tile
    unsigned int tilePixelPosCol = threadIdx.x;
    // Input image pixel column position
    unsigned int iPixelPosCol = tileStartCol + tilePixelPosCol;

    unsigned int tilePixelPosRow = 0;
    unsigned int iPixelPosRow = 0;
    unsigned int iPixelPos = 0;
```

```

unsigned int tilePixelPos = 0;

for (int subBlockNo = 0; subBlockNo < noSubBlocks; subBlockNo++) {
    tilePixelPosRow = threadIdx.y + subBlockNo * blockDim.y;
    iPixelPosRow = tileStartRow + tilePixelPosRow;

    // Check if the pixel is in the image
    if (iPixelPosCol < tileEndClampedCol && iPixelPosRow < tileEndClampedRow) {
        iPixelPos = iPixelPosRow * paddedWidth + iPixelPosCol;
        tilePixelPos = tilePixelPosRow * tileSize + tilePixelPosCol;
        // Load the pixel in the shared memory
        s_data[tilePixelPos] = d_sourceImagePtr[iPixelPos];
    }
}

// Wait for threads loading data in tiles
__syncthreads();

unsigned int oPixelPosRow = 0;
unsigned int oPixelPos = 0;
unsigned int tilePixelPosOffset = 0;
unsigned int maskIndex = 0;
unsigned int oPixelPosCol = iPixelPosCol - surroundingPixels;

for (int subBlockNo = 0; subBlockNo < noSubBlocks; subBlockNo++) {

    float pixelSum = 0;
    tilePixelPosRow = threadIdx.y + subBlockNo * blockDim.y;
    iPixelPosRow = tileStartRow + tilePixelPosRow;

    // Check if the pixel is in the tile and image.
    // Pixels in the tile padding are exclude from evaluation.
    if (iPixelPosCol >= tileStartCol + surroundingPixels &&
        iPixelPosCol < tileEndClampedCol - surroundingPixels &&
        iPixelPosRow >= tileStartRow + surroundingPixels &&
        iPixelPosRow < tileEndClampedRow - surroundingPixels) {

        // Evaluate pixel position for output image
        oPixelPosRow = iPixelPosRow - surroundingPixels;
        oPixelPos = oPixelPosRow * width + oPixelPosCol;

        tilePixelPos = tilePixelPosRow * tileSize + tilePixelPosCol;

        // Apply convolution
        for (int h = -surroundingPixels; h <= surroundingPixels; h++) {
            for (int w = -surroundingPixels; w <= surroundingPixels; w++) {
                tilePixelPosOffset = h * tileSize + w;
                maskIndex = (h + surroundingPixels) * filterWidth + (w +
                    surroundingPixels);
                pixelSum += s_data[tilePixelPos + tilePixelPosOffset] *
                    d_cFilterKernel[maskIndex];
            }
        }

        // Thresholding overflowing pixel's values
        if (pixelSum < 0) {
    }
}

```

```

        pixelSum = 0;
    }

    else if (pixelSum > 255) {
        pixelSum = 255;
    }

    // Write pixel on the output image
    d_outImagePtr[oPixelPos] = pixelSum;
}
}

```

I pixel relativi ad un blocco 32x32 possono essere caricati nella shared memory una sola volta e riutilizzati dai thread all'interno dello stesso blocco. Questo riduce la necessità di caricare ripetutamente gli stessi dati dalla memoria globale, migliorando le prestazioni complessive.

2.2 Risultati dei test

La metrica utilizzata per confrontare le prestazioni dell'algoritmo sequenziale con le versioni parallele in Java e CUDA è lo speedup, calcolato come:

$$S = \frac{t_S}{t_P}$$

dove t_S e t_P sono rispettivamente il tempo di esecuzione della versione sequenziale e di quella parallela dell'algoritmo. I dataset utilizzati per le diverse implementazioni sono stati generati con la funzione `make_blob` di `sklearn.datasets`. [5]

Il test è stato effettuato sulla stessa immagine in 3 qualità diverse (480p, FullHD, 4K). Sono stati applicati gli stessi kernel in dimensioni diverse (3x3, 5x5 e 7x7) e per ogni combinazione di queste caratteristiche sono stati misurati i tempi di esecuzione per la versione sequenziale e per quelle parallele.

I test sono stati eseguiti su un processore AMD Ryzen 5 3500U, 4 core (8 logici). Come possiamo vedere, le prestazioni dell'operazione di convoluzione aumentano con l'aumentare del numero di thread e i miglioramenti delle

prestazioni sono più evidenti all'aumentare delle dimensioni del kernel e della risoluzione delle immagini.

Nella tabella 2.1 sono riportati i risultati dei tempi di esecuzione, e lo speedup nel caso in di immagine 4K, alla quale applichiamo un filtro 7x7, al variare del numero di thread.

		Tempo sequenziale	14,192
	Threads	Time (seconds)	Speedup
	2	7,854	1,81
	3	6,101	2,33
	4	5,112	2,78
	5	4,528	3,13
	6	4,376	3,24
	7	4,486	3,16
	8	4,550	3,18

Table 2.1: Speedup con immagine 4K, filtro 7x7, al variare del numero di thread

In figura 2.3 vediamo come lo speedup aumenti all'aumentare del numero dei thread.

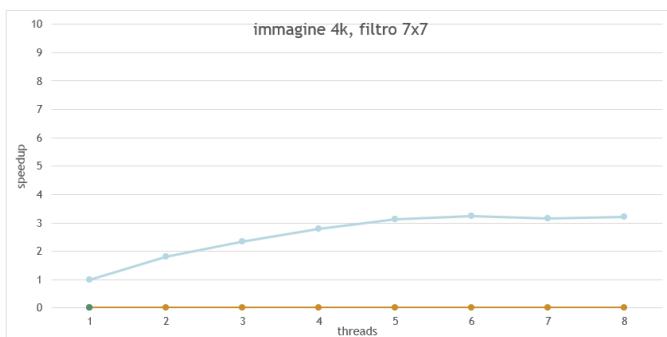


Figure 2.3: Speedup con immagine 4K al variare del numero det hread con Java

Nelle tabelle 2.2, 2.3, 2.4 sono riportati i risultati dei tempi di esecuzione,

480p(s)			
mask	sequenziale	parallelo	speedup
3x3	0,860	0,172	5,00
5x5	1,040	0,195	5,33
7x7	1,171	0,215	5,45

Table 2.2: Speedup con immagine 480p al variare della dimensione del filtro nel caso di pixel suddivisi per colonne tra i vari thread

Full HD(s)			
mask	sequenziale	parallelo	speedup
3x3	0,910	0,192	4,74
5x5	1,303	0,265	4,92
7x7	1,619	0,328	4,94

Table 2.3: Speedup con immagine Full HD al variare della dimensione del filtro nel caso di pixel suddivisi per colonne tra i vari thread

4k(s)			
mask	sequenziale	parallelo	speedup
3x3	5,837	1,114	5,24
5x5	8,997	1,725	5,22
7x7	16,035	2,908	5,51

Table 2.4: Speedup con immagine 4K al variare della dimensione del filtro nel caso di pixel suddivisi per colonne tra i vari thread

nel caso in cui l'immagine venga suddivisa per colonne.

Nelle tabelle 2.5, 2.6, 2.7 sono riportati i risultati dei tempi di esecuzione, nel caso in cui l'immagine venga suddivisa per righe.

480p(s)			
mask	sequenziale	parallelo	speedup
3x3	0,801	0,166	4,83
5x5	1,104	0,213	5,18
7x7	1,183	0,237	4,99

Table 2.5: Speedup con immagine 480p al variare della dimensione del filtro nel caso di pixel suddivisi per righe tra i vari thread

Full HD(s)			
mask	sequenziale	parallelo	speedup
3x3	0,931	0,199	4,68
5x5	1,223	0,271	4,51
7x7	1,673	0,375	4,46

Table 2.6: Speedup con immagine Full HD al variare della dimensione del filtro nel caso di pixel suddivisi per righe tra i vari thread

4k(s)			
mask	sequenziale	parallelo	speedup
3x3	5,809	1,104	5,26
5x5	8,868	1,628	5,45
7x7	16,221	2,905	5,58

Table 2.7: Speedup con immagine 4K al variare della dimensione del filtro nel caso di pixel suddivisi per righe tra i vari thread

Nelle tabelle 2.8 e 2.9 sono riportati i risultati dei tempi di esecuzione dell'applicazione in CUDA con e senza considerare il trasferimento dei dati da CPU a GPU, e lo speedup nel caso di immagine 4K, al variare della dimensione del filtro applicato.

mask	tempo esecuzione filtro (seconds)	tempo esecuzione		
		Speedup	filtro + trasferimento dati (seconds)	Speedup
3x3	0,000849	6842,17	0,160506	36,19
5x5	0,001523	5822,72	0,164288	53,98
7x7	0,002494	6504,01	0,183203	88,54

Table 2.8: Speedup con immagine 4K, al variare della dimensione del filtro con CUDA

mask	tempo esecuzione filtro (seconds)	tempo esecuzione		
		Speedup	filtro + trasferimento dati (seconds)	Speedup
3x3	0,000480	12102,08	0,164831	35,24
5x5	0,000919	9649,62	0,157939	56,15
7x7	0,001459	11117,89	0,161872	100,21

Table 2.9: Speedup con immagine 4K, al variare della dimensione del filtro con CUDA, versione con utilizzo shared e cstant memory

Infine, nelle tabelle 2.8 e 2.9 sono riportati i risultati della versione dell'applicazione sviluppata in CUDA. Sono mostrati i tempi comprensivi del tempo di trasferimento tra CPU e GPU ed i tempi della sola applicazione del filtro da parte del kernel della GPU. Si nota come lo speedup raggiunga valori molto alti, mostrando come l'utilizzo di CUDA sia ottimale per questo tipo di operazioni su immagini.

Chapter 3

Conclusioni

In questo lavoro è stato presentato l'algoritmo di clustering Mean Shift ed è stato mostrato come la sua struttura imbarazzantemente parallela lo renda adatto per il calcolo parallelo. È stata sviluppata un'implementazione parallela con OpenMP aggiungendo soltanto una direttiva che ha permesso di ottenere un aumento dello speedup di oltre 5. Successivamente è stata presentata un'implementazione in CUDA e con i suoi tempi di esecuzione ha mostrato come l'utilizzo della GPU renda l'algoritmo di Mean Shift applicabile anche a set di dati con dimensioni intrattabili per le CPU.

Bibliography

- [1] kernel. https://it.wikipedia.org/wiki/Matrice_di_convoluzione.
- [2] guida cuda. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.
- [3] constant memory. <https://devtalk.nvidia.com/default/topic/910290/using-constant-memory/>.
- [4] shared memory. <https://devblogs.nvidia.com/using-shared-memory-cuda-cc/>.
- [5] F. pedregosa, g. varoquaux, a. gramfort, v. michel, b. thirion, o. grisel, m. blondel, p. prettenhofer, r. weiss, v. dubourg, j. Vanderplas, a. passos, d. cournapeau, m. brucher, m. perrot, and e. duchesnay. scikit-learn: Machine learning in python. *journal of machine learning research*, 12:2825–2830, 2011. https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html.