Parallel Image Composition

Elaborato Parallel Programming For Machine Learning

Nicoló Pollini, Francesco Fantechi

A.A. 2022-2023

UNIVERSITA' DEGLI STUDI DI FIRENZE
Facolta di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Contents

1 Obiettivo

L'obiettivo di questo progetto è quello di parallelizzare un algoritmo di Data Augmentation che effettua composizioni di immagini tramite il framework OpenMP e le librerie di Python che permettono il multiprocessing e confrontare i vari metodi valutando gli speedup ottenuti rispetto alle loro esecuzioni sequenziali.

2 Data Augmentation

La Data Augmentation é un insieme di processi atti ad aumentare gli elementi di un dataset senza raccogliere nuovi dati. Queste operazioni consistono infatti nell'effettuare cambiamenti controllati agli elementi del dataset ottenendo così copie modificate dei dati già essitenti. Questa tecnica trova un largo impiego nell'addestramento delle reti neurali quando i dati a disposizione non sono numericamente sufficienti e raccoglierne di nuovi risulterebbe troppo costoso o addirittura infattibile.
Nel nostro esperimento i dati da aumentare corrispondono a delle immagini raffiguranti dei quokka e per ottenerne di nuovi é stato effettuato un processo di Image Composition. Partendo da cinque immagini di background di dimensione 483×594 pixel, sono state infatti generate nuove immagini andando ad applicare sopra di esse in punto randomico un'ulteriore immagine di dimensione 298×350 pixel dopo avergli applicato una trasparenza anch'essa casuale. (vedi figura 1)

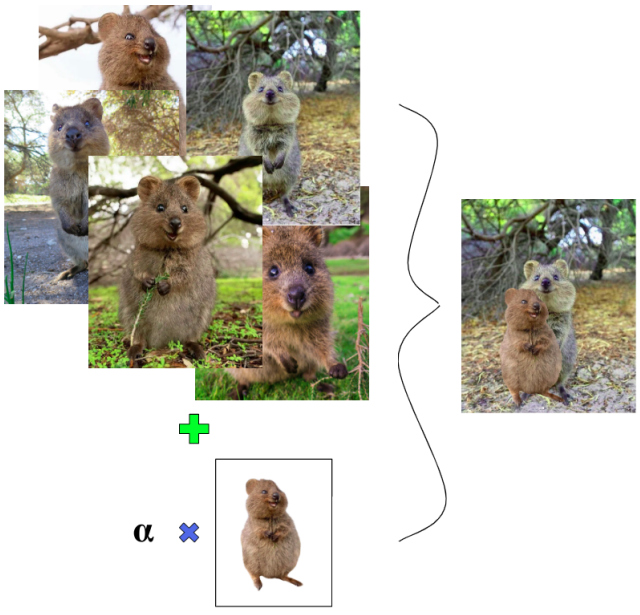

Figure 1: Data Augmentation mediante Image Composition

3 Codice e implementazione

Il progetto è stato implementato nei linguaggi C++ e Python ed é stato eseguito su due diverse macchine le cui caratteristiche sono riportate in Tabella 1.
Il codice é stato versionato tramite la piattaforma GitHub ed é reperibile ai siti:
https://github.com/francesco-ftk/Image_Composition_OpenMP.git
https://github.com/francesco-ftk/Image_Composition_Multiprocessing.git

Il progetto puó essere suddiviso in tre parti:

1. Implemenatazione dell'agoritmo di Image Composition sequenziale
2. Parallelizzazione dell'algoritmo tramite il framework OpenMP in C++
3. Parallelizzazione dell'algoritmo tramite le librerie di multiprocessing in Python

3.1 Image Composition Sequenziale

Date le immagini di partenza e un parametro *"transformations"* indicativo del numero di nuove immagini volute, l'Image Composition é stata eseguita andando a scegliere ad ogni nuova trasformazione un'immagine fra quelle possibili di background in modo randomico. A queste é stata quindi aggiunta in posiszione randomica l'immagine di foreground con una trasparenza uniformemente scelta nell'intervallo $[128, 255]$. Le immagini sono state lette, editate e salvate tramite la libreria OpenCV. In figura 2 é riporato un estratto del codice sequenziale che implementa l'Image Composition scritto in linguaggio Python. Nonostante la sintassi diversa, la versione in scritta in linguaggio C++ si presenta praticamente uguale per quanto riguarda la forma.
I tempi di completamento in secondi per il codice scritto in linguaggio Python e in millisecondi per il codice scritto in C++ sono riportati in tabella 2 e 3.

```python
def data_augmentation(foreground, backgrounds, transformations) -> int:
    # generate the directory for saving ...
    for i in range(0, transformations, 1):
        index = random.randint(0, len(backgrounds) - 1)
        background = copy(backgrounds[index])
        # check the dimensions between foreground and background ...
        if background.shape[0] - foreground.shape[0] == 0:
            row = 0
        else:
            row = random.randint(0, background.shape[0] - foreground.shape[0] - 1)
        if background.shape[1] - foreground.shape[1] == 0:
            col = 0
        else:
            col = random.randint(0, background.shape[1] - foreground.shape[1] - 1)
        alpha = random.randint(128, 255)
        alpha_correction_factor = float(alpha) / 255.0
        beta_correction_factor = 1.0 - alpha_correction_factor
        for j in range(0, foreground.shape[0], 1):
            for k in range(0, foreground.shape[1], 1):
                f_pixel = foreground[j, k]
                b_pixel = background[row + j, col + k]
                f_alpha = float(f_pixel[3]) / 255.0
                if f_alpha > 0.9:
                    background[row + j, col + k] = [float(b_pixel[0]) * beta_correction_factor + float(
                        f_pixel[0]) * alpha_correction_factor * f_alpha,
                                                    float(b_pixel[1]) * beta_correction_factor + float(
                                                        f_pixel[1]) * alpha_correction_factor * f_alpha,
                                                    float(b_pixel[2]) * beta_correction_factor + float(
                                                        f_pixel[2]) * alpha_correction_factor * f_alpha, 255]
        # save the new image ...
```

Figure 2: Codice Python che implementa l'algoritmo di Image Composition

3.2 OpenMP

OpenMP é un framework che permette di parallelizzare una porzione di codice in modalitá implicit threading attraverso delle direttive "pragma".
Nel nostro esperimento la tipologia di parallelizzazione utilizzata é stata di tipo coarse-grained, ossia anziché parallelizzare il processo stesso di composizione delle immagini, viene parallelizzato il ciclo esterno. In altre parole, ogni thread genera una porzione delle immagini di output richieste tramite il parametro *"transformations"*, in modo che ogni nuova immagine venga ottenuta in modo indipendente dalle altre. L'algoritmo risuta quindi imbarazzantemente parallelo e quindi facilmente parallelizzabile in quanto i thread non necessitano di comunicare o scambiare informazioni fra di loro (Vedi figura 3).
Il numero di thread usati é stato calcolato aggiungendo un 50% al numero complessivo di Core posseduti dalla macchina utilizzata considerando l'Hyper-Threading (vedi Tabella 1). I tempi di esecuzione in millisecondi della Macchina 1 (vedi Tabella 1) presi in modialitá "Debug" sono riportati in Tabella 3. Con questi tempi t_p e quelli presi tramite l'implementazione sequenziale t_s sono stati successivamente calcolati gli speedup tramite la formula $S = t_s/t_p$. Questi ultimi sono osservabili in Tabella 4.

In Figura 7 é riportato un grafico che mostra i tempi di esecuzione in millisecondi per la Macchina 1 (vedi Tabella 1) al variare del numero dei thread per genereare 1800 immagini.
In Figura 8 é possibile osservare invece un grafico che mostra gli speedups ottenuti dalla Macchina 1 (vedi Tabella 1) al variare del numero di immagini generate utilizzando 18 thread.

```
#pragma omp parallel default(none) shared(foreground, backgrounds) firstprivate(transformations, outputFolderPath)
{
    #pragma omp for
    for (int count = 0; count < transformations; ++count) {
        // copy a random background from the shared pool
        int index = (int) (uniform(gen) % backgrounds.size());
        cv::Mat background = backgrounds.at(index).clone();
        // pick a random position for the foreground
        int row;
        if (background.rows - foreground.rows == 0) { row = 0; }
        else { row = (int) (uniform(gen) % (background.rows - foreground.rows)); }
        int col;
        if (background.cols - foreground.cols == 0) { col = 0; }
        else { col = (int) (uniform(gen) % (background.cols - foreground.cols)); }
        // pick a random alpha value
        int alpha = 128 + uniform(gen) % 128;
        float alphaCorrectionFactor = (float) alpha / 255;
        float betaCorrectionFactor = 1 - alphaCorrectionFactor;

        for (int i = 0; i < foreground.rows; ++i) {
            for (int j = 0; j < foreground.cols; ++j) {
                cv::Vec4b &f_pixel = foreground.at<cv::Vec4b>(i, j);
                cv::Vec4b &b_pixel = background.at<cv::Vec4b>(row + i, col + j);
                float f_alpha = (float) f_pixel[3] / 255;
                if (f_alpha > 0.9) {
                    b_pixel[0] = (unsigned char) ((float) b_pixel[0] * betaCorrectionFactor +
                                                  (float) f_pixel[0] * alphaCorrectionFactor * f_alpha);
                    b_pixel[1] = (unsigned char) ((float) b_pixel[1] * betaCorrectionFactor +
                                                  (float) f_pixel[1] * alphaCorrectionFactor * f_alpha);
                    b_pixel[2] = (unsigned char) ((float) b_pixel[2] * betaCorrectionFactor +
                                                  (float) f_pixel[2] * alphaCorrectionFactor * f_alpha);
                }
            }
        }
        std::string outputPath = outputFolderPath + "/out_" + std::to_string(count) + ".png";
        cv::imwrite(outputPath, background);
    }
}
```

Figure 3: Parallelizzazione tramite framework OpenMP in C++

3.3 Multiprocessing

A causa dell'interprete GIL (Global Interpretr Lock) che gestisce il garbage collector con dei contatori di riferimento sugli oggetti per determinare quando non sono piú in uso, in Python non é possibile eseguire piú di un thread contemporaneamente. Un modo per superare queste limitazioni di concorrenza puó essere quello di istanziare dei sottoprocessi al posto dei thread. Ogni processo possederá infatti il proprio GIL e potrá essere eseguito in modo asincrono su un

core diverso ottenendo cosí il parallelismo. Il superamento delle restrinzioni di concorrenza va peró a discapito della performance in quanto istanziare i processi risulta molto piú costoso ripetto a far partire dei thread.
Nel nostro esperimento sono stati implementati due metodi differenti per istanziare i sottoprocessi: il primo basato sulla libreria standard di Python Multiprocessing e il secondo tramite la libreria Joblib.
La libreria Multiprocessing permette di istanziare e far partire manualmente i processi su una certa funzione target passata alla loro creazione. Il lavoro é stato equamente suddiviso fra i vari processi. Ognuno di essi infatti riceve come argomenti le immagini di background e foreground e il parametro *"transformations_for_process"* ottenuto dividendo il numero di trasformazioni volute per il numero di processi istanziati. Due implementazioni equivalenti del suo utilizzo sono riportate in figura 4 e 5.
In Figura 9 é possibile osservare un grafico che mostra gli speedups ottenuti dalla Macchina 2 (vedi Tabella 1) al variare del numero di immagini generate utilizzando 12 processi.

```
# MULTIPROCESSING IMAGE COMPOSITION
print("START Pool Multiprocessing Algorithm")

# pool_size = multiprocessing.cpu_count() * 2
pool_size = PROCESSES
print("Using " + str(pool_size) + " processes")
pool = multiprocessing.Pool(
    processes=pool_size,
)
trasformations_for_process = math.ceil(TRANSFORMATIONS / pool_size)
start = time.time()
local_date = datetime.datetime.now()
new_dir_path = 'output/' + str(local_date)
os.mkdir(new_dir_path)
# prepare arguments iterable for pool.starmap
args = [(foreground, backgrounds, new_dir_path, trasformations_for_process) for i in range(pool_size)]
pool.starmap(data_augmentation_multiprocessing, args)

pool.close()  # no more tasks
pool.join()  # wrap up current tasks

end = time.time()
print(f'Pool Multiprocessing Running took {end - start} seconds.')
```

Figure 4: Codice Python che utilizza la librearia Multiprocessing con Pooling

```
print("START Multiprocessing Algorithm")
print("Using " + str(PROCESSES) + " processes")

trasformations_for_process = math.ceil(TRANSFORMATIONS / PROCESSES)
local_date = datetime.datetime.now()
new_dir_path = 'output/' + str(local_date)
os.mkdir(new_dir_path)
processes = [Process(target=data_augmentation_multiprocessing,
                     args=(foreground, backgrounds, new_dir_path, trasformations_for_process,)) for i in
             range(PROCESSES)]
start = time.time()
[p.start() for p in processes]
[p.join() for p in processes]
end = time.time()

print(f'Multiprocessing Running took {end - start} seconds.')
```

Figure 5: Codice Python che utilizza la libreria Multiprocessing

La libreria Joblib permette di parallelizzare mediante il multiprocessing programmi imbarazzantemente paralleli. I processi vengono eseguiti in modo asincrono sulla propria funzione target suddividendosi il lavoro in modo autonomo. Dato che ad ogni nuova esecuzione di un processo gli argomenti posseduti devono essere ripassati in ingresso alla funzione, per ridurre i costi ogni processo genera una nuova immagine alla volta e riceve giá negli argomenti l'immagine di background da utilizzare e non l'intero dataset da cui scegliere. Il codice che implementa questa libreria é riportato in figura 6.

```
# JOBLIB MULTIPROCESSING IMAGE COMPOSITION
print("START Multiprocessing Joblib Algorithm")

start = time.time()
local_date = datetime.datetime.now()
new_dir_path = 'output/' + str(local_date)
os.mkdir(new_dir_path)
index = random.randint(0, len(backgrounds) - 1)
background = backgrounds[index]
Parallel(n_jobs=PROCESSES)(
    delayed(data_augmentation_multiprocessing_one_at_a_time)(foreground, background, new_dir_path, i) for i
    in range(TRANSFORMATIONS))
end = time.time()

print(f'Multiprocessing Joblib Running took {end - start} seconds.')
```

Figure 6: Codice Python che utilizza la libreria Joblib

Il numero di processi istanziati é stato calcolato aggiungendo un 50% al numero complessivo di Core posseduti dalla macchina utilizzata considerando l'Hyper-Threading (vedi Tabella 1). I tempi di completamento in secondi della Macchina 2 (vedi Tabella 1) sono riportati in Tabella 2. Con questi tempi sono stati successivamente calcolati gli speedup di entrambi i metodi rispetto all'esecuzione sequenziale. Questi ultimi sono osservabili in Tabella 4.

4 Risultati

Table 1: Machine's technical specifications

Machine 1			
OS	CPU	Number of Core with Hyper-Threading	RAM
Windows 10	Intel(R) Core(TM) i7-8750H	12	16 GB
Machine 2			
OS	CPU	Number of Core with Hyper-Threading	RAM
Ubuntu 20.04.5	Intel(R) Core(TM) i7-1165G7	8	16 GB

Table 2: Python Timings

Machine 2				
Number of Images	Sequential	Joblib	Pool Multiprocessing	Multiprocessing
1200	$122.1s$	$143.9s$	$52.1s$	$50.7s$

Table 3: C++ Timings

Machine 1		
Number of Images	Sequential	OpenMP
1800	$88974ms$	$14832ms$

Table 4: Speedups

Machine 1		
OpenMP		
6.0		
Machine 2		
Joblib	Pool Multiprocessing	Multiprocessing
0.9	2.3	2.4

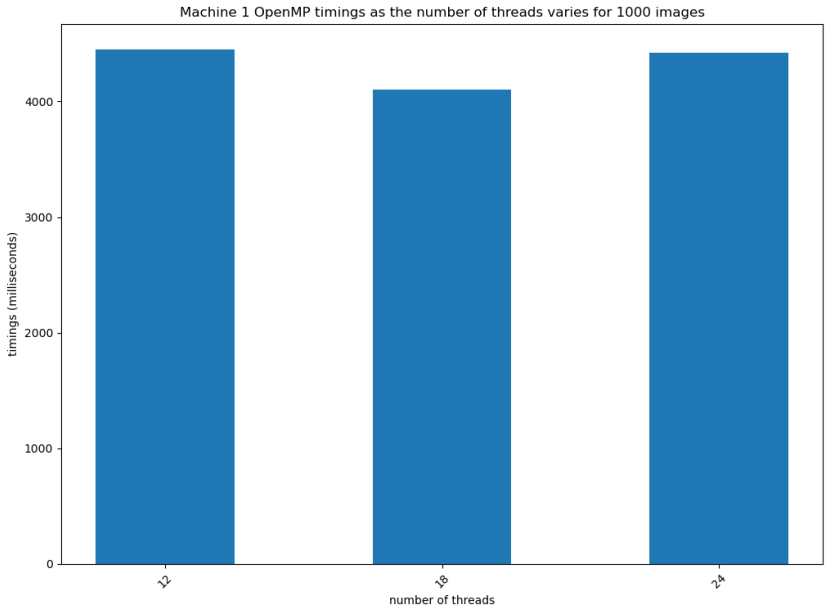

Figure 7: Machine 1 OpenMP timings as the number of threads varies

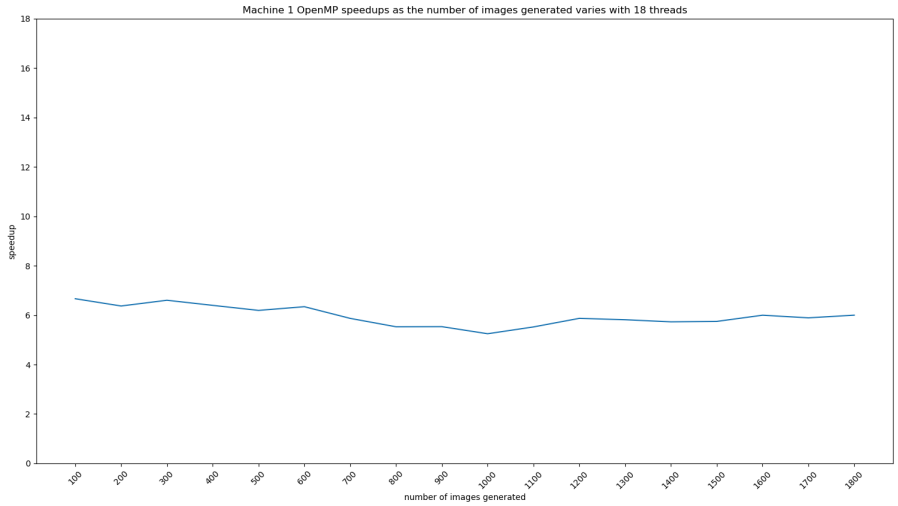

Figure 8: Machine 1 OpenMP speedups as the number of generated images varies

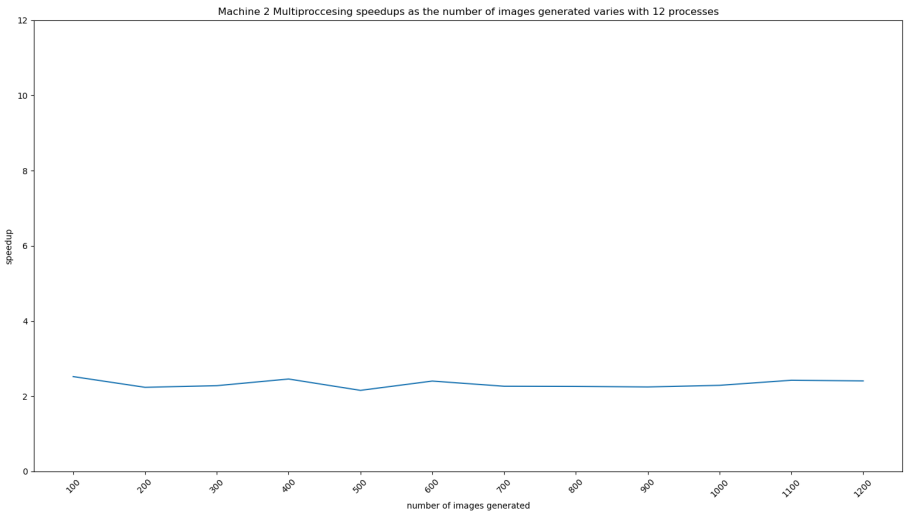

Figure 9: Machine 2 Multiprocessing speedups as the number of generated images varies

5 Analisi e Conclusioni

Come previsto, la parallelizzazione ha permesso di ottenere uno speedup sublineare, che nel caso di OpenMP é molto vicino al numero di core fisici presenti nella macchina utilizzata. Questo risultato può essere motivato grazie ad una parallelizzazione effettuata dividendo il carico di lavoro in modo equo sui thread disponibili. Se il numero di immagini da generare fosse stato troppo basso sarebbe stato possibile osservare una diminuzione delle performance di parallelizzazione a causa di uno sbilanciamento del carico di lavoro sui thread; tuttavia, come é possibile apprezzare dal grafico 8, se il numero di trasformazioni rimane abbastanza alto, lo speedup rimane invece stabile al variare del carico di lavoro.
Lo stesso comportamento é apprezzabile in figura 9 per la parallelizzazione in Python tramite la la libreria di Multiprocessing. In questo caso peró, essendo i processi più costosi da far partire rispetto ai thread e possedendo la Macchina 2 (vedi Tabella 1) un numero inferiore di core, lo speedup risulta nettamente inferiore a quello ottenuto con OpenMP.
Come é possibile osservare in Tabella 4, lo speedup risulta inesistente invece utilizzando la libreira Joblib forse per il costo maggiore dei processi e il modo di instanziarli nuovamente di quest'ultima ad ogni nuovo task.