Relazione Progetto CUDA

Silviu Robert Plesoiu - 7051276

September 5, 2024

1 Introduzione

Il progetto qui descritto riguarda l'implementazione di una convoluzione 2D utilizzando CUDA per l'elaborazione di immagini. L'obiettivo principale del progetto è sfruttare la parallelizzazione offerta dalla GPU per migliorare le prestazioni rispetto a un'implementazione sequenziale su CPU. La tecnologia utilizzata include CUDA per la parallelizzazione e OpenCV per la gestione delle immagini.

2 Algoritmo

L'algoritmo implementato esegue una convoluzione 2D su un'immagine utilizzando un kernel definito dall'utente. Il kernel viene applicato separatamente ai tre canali dell'immagine (rosso, verde e blu). La convoluzione viene eseguita su ciascun canale in parallelo sulla GPU, sfruttando i blocchi e i thread per processare i pixel in parallelo.

Le principali regole dell'algoritmo sono:

- Per ogni pixel dell'immagine, il kernel viene applicato considerando un'area di dimensione 3×3 intorno al pixel.
- I bordi dell'immagine vengono gestiti tramite il clamping dei valori di indice (usando funzioni custom min e max).
- La memoria condivisa viene utilizzata per ridurre il numero di accessi alla memoria globale.

2.1 Pseudocodice

```
per ogni pixel (x, y) dell'immagine:
    somma = 0
    per ogni valore nel kernel (i, j):
        pixel_correlato = immagine[x+i, y+j]
        somma += pixel_correlato * kernel[i, j]
    output[x, y] = somma
```

3 Implementazione Tecnica

3.1 Librerie Utilizzate

- **CUDA**: utilizzato per eseguire il calcolo parallelo della convoluzione.
- **OpenCV**: utilizzato per la gestione delle immagini, come il caricamento e il salvataggio, e la conversione dei canali.

3.2 Parallelizzazione con CUDA

Il codice CUDA sfrutta una griglia di blocchi e thread per parallelizzare l'applicazione del kernel di convoluzione. Ogni thread gestisce un singolo pixel dell'immagine e applica il kernel a quel pixel.

Il codice seguente mostra l'implementazione del kernel CUDA per la convoluzione 2D:

```
1  __global__ void convoluzione2D(float* __restrict__
       immagine, float* __restrict__ output, float*
       __restrict__ kernel, int larghezza, int altezza) {
2      extern __shared__ float shared_mem[];
3      float* shared_image = shared_mem;
4      float* shared_kernel = shared_mem + (BLOCK_SIZE + 2)
           * (BLOCK_SIZE + 2);
5
6      int x = blockIdx.x * blockDim.x + threadIdx.x;
7      int y = blockIdx.y * blockDim.y + threadIdx.y;
8
9      int tx = threadIdx.x;
10     int ty = threadIdx.y;
11
12     int kernel_radius = N / 2;
13
14     if (tx < N && ty < N) {
```

```
15          shared_kernel[ty * N + tx] = kernel[ty * N + tx
                ];
16      }
17
18      __syncthreads();
19
20      int xk = custom_min(custom_max(x - kernel_radius, 0)
            , larghezza - 1);
21      int yk = custom_min(custom_max(y - kernel_radius, 0)
            , altezza - 1);
22
23      shared_image[(ty + kernel_radius) * (BLOCK_SIZE + 2)
            + (tx + kernel_radius)] = immagine[yk *
            larghezza + xk];
24
25      if (tx < kernel_radius) {
26          shared_image[ty * (BLOCK_SIZE + 2) + tx] =
                immagine[yk * larghezza + custom_max(xk -
                kernel_radius, 0)];
27      }
28      if (ty < kernel_radius) {
29          shared_image[ty * (BLOCK_SIZE + 2) + tx +
                kernel_radius] = immagine[custom_max(yk -
                kernel_radius, 0) * larghezza + xk];
30      }
31      if (tx >= blockDim.x - kernel_radius) {
32          shared_image[ty * (BLOCK_SIZE + 2) + tx + 2 *
                kernel_radius] = immagine[yk * larghezza +
                custom_min(xk + kernel_radius, larghezza - 1)
                ];
33      }
34      if (ty >= blockDim.y - kernel_radius) {
35          shared_image[(ty + 2 * kernel_radius) * (
                BLOCK_SIZE + 2) + tx + kernel_radius] =
                immagine[custom_min(yk + kernel_radius,
                altezza - 1) * larghezza + xk];
36      }
37
38      __syncthreads();
39
40      if (x >= larghezza || y >= altezza) return;
41
42      float valore = 0.0;
```

```
43
44      for (int i = 0; i < N; i++) {
45          for (int j = 0; j < N; j++) {
46              valore += shared_image[(ty + i) * (
                    BLOCK_SIZE + 2) + (tx + j)] *
                    shared_kernel[i * N + j];
47          }
48      }
49
50      output[y * larghezza + x] = valore;
51  }
```

4 Benchmark e Risultati

Il test delle prestazioni è stato eseguito utilizzando un kernel fisso di dimensioni 3×3. Il focus è stato sul confronto tra una versione semplice, che non sfrutta ottimizzazioni particolari, e una versione ottimizzata con l'uso della memoria condivisa.

4.1 Risultati di Esecuzione

I tempi di esecuzione ottenuti sono riportati nella seguente tabella:

Versione	**Tempo di esecuzione (ms)**
Versione semplice	18.68 ms
Versione ottimizzata con memoria condivisa	0.098 ms

Table 1: Confronto tra versione semplice e ottimizzata.

4.2 Spiegazione delle prestazioni

Il miglioramento delle prestazioni nella versione ottimizzata è dovuto principalmente a due fattori:

- **Uso della memoria condivisa**: La versione ottimizzata sfrutta la memoria condivisa per immagazzinare i dati del kernel e una porzione dell'immagine che viene convoluta. Questo riduce drasticamente il numero di accessi alla memoria globale della GPU, che è molto più lenta rispetto alla memoria condivisa. Riducendo gli accessi alla memoria globale, i thread possono accedere ai dati in maniera più efficiente e parallela, con un impatto positivo sulle prestazioni complessive.

- **Sincronizzazione dei thread**: La sincronizzazione dei thread tramite `__syncthreads()` è essenziale per garantire che tutti i dati necessari siano caricati nella memoria condivisa prima che i calcoli inizino. Questo evita condizioni di corsa e garantisce che ogni thread abbia accesso ai dati corretti, migliorando l'accuratezza e l'efficienza dell'esecuzione parallela.

In sintesi, la combinazione dell'uso efficiente della memoria condivisa e di una corretta gestione dei thread ha permesso di ottenere un miglioramento drastico nelle prestazioni. Questi risultati dimostrano l'importanza di comprendere e applicare correttamente le tecniche di ottimizzazione in CUDA, come trattato nel corso.