Integral Image CUDA

Giovannoni Alberto

Abstract

In questa relazione, esploriamo l'implementazione di Integral Image utilizzando CUDA, un linguaggio per la programmazione parallela. L'obiettivo principale è confrontare le prestazioni fra versione sequenziale e parallela.

1. Introduzione

1.1. Problema di integral image

In questo elaborato si vogliono confrontare le prestazioni fra versione sequenziale di integral image e versione parallela implementata in CUDA. Prima di fare questo però è importante capire cosa sia la integral image di una matrice. Nell'integral image, il valore in posizione (i,j) è ottenuto sommando tutti i valori contenuti nella matrice che va dall'origine al valore in (i,j) stesso. Di seguito, in figura 1, è riportato un esempio.

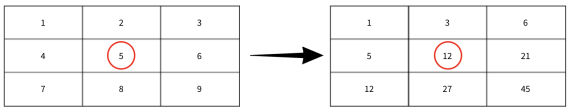

Figure 1. Esempio del calcolo dell'integral image

Per ottenere ad esempio il valore 12, si devono sommare 1, 2, 4 e 5.

1.2. Calcolare l'integral image

In letteratura esistono varie tecniche per il calcolo dell'integral image, ma non tutte si prestano bene al calcolo parallelo. Difatti uno dei metodi possibili è quello di calcolare l'elemento in posizione (i,j) sfruttando i risultati degli elementi in posizione (i-1, j), (i, j-1) e (i-1, j-1). Ad esempio riferendosi alla figura 1, si potrebbe calcolare 45 come 21 + 27 + 9 - 12. Questo tipo di soluzione è impraticabile in una versione parallela, in quanto i vari thread dovrebbero attendere il calcolo dei propri vicini, introducendo una considerevole sincronizzazione. Per ovviare a questo problema, è stato pubblicato un articolo dall'Università della California [1] che mostra come si possa calcolare l'integral image a partire da operazioni di scan e transpose. In pratica vengono eseguite 4 operazioni:

- scan delle righe della matrice, ogni elemento è ottenuto sommando quelli che lo precedono
- transpose della matrice, in questo modo le righe e le colonne vengono scambiate
- scan sulla matrice trasposta, questo ha come risultato l'integral image trasposta
- traspose, per ottenere l'integral image definitiva

In figura 2 è riportato un esempio.

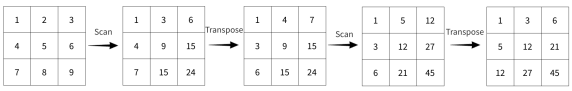

Figure 2. Esempio del calcolo dell'integral image con scan e transpose

Il vantaggio di questa soluzione sta nel fatto che il calcolo di scan e transpose è completamente parallelizzabile. Nell'elaborato la versione parallela verrà confrontata prima con una versione sequenziale di scan-transpose, e poi con la versione

sequenziale ottima che sfrutta i risultati parziali vicini nella matrice.

1.2.1 Scan

Come algoritmo di scan si usa una versione che funziona tramite la costruzione di un albero bilanciato delle somme parziali del vettore (come si può vedere dalla figura 3). Per fare questo sono necessarie due fasi, chiamate up-sweep e down-sweep. Nella fase di up-sweep i valori del vettore vengono combinati per calcolare le somme parziali, mentre in quella di down-sweep, le somme parziali vengono distribuite all'interno dell'array, completando così il calcolo della scansione. Nelle figure 4 e 5 è riportato un esempio.

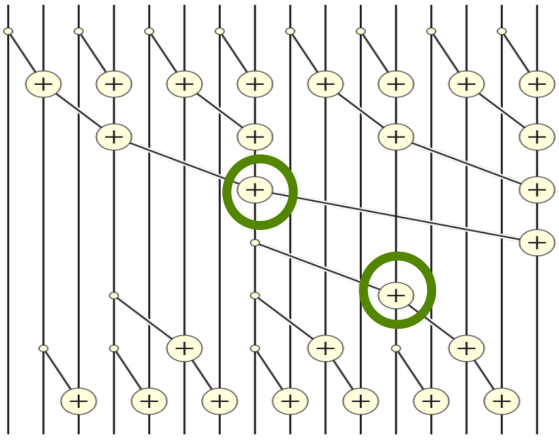

Figure 3. Rappresentazioni ad albero binario di up-sweep e down-sweep

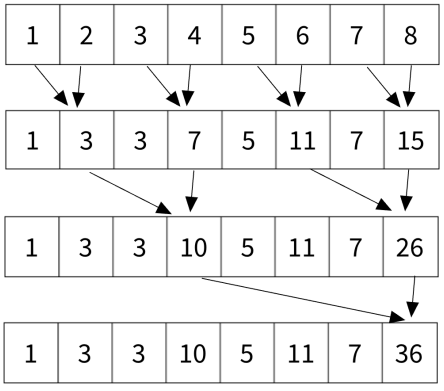

Figure 4. Fase di up-sweep

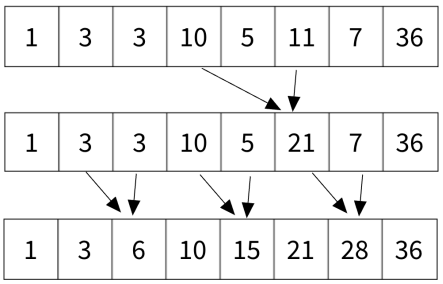

Figure 5. Fase di down-sweep

2. Struttura del codice

All'interno del progetto sono contenuti i seguenti file:

- **sequential.cpp:** contiene le versioni sequenziali (scan-transpose e ottimizzata) di integral image
- **parallel.cu:** contiene la versione parallela CUDA di integral image
- **scan.cu:** contiene il codice CUDA del metodo scan parallelizzato
- **transpose.cu:** contiene il codice CUDA del metodo transpose parallelizzato

2.1. Creazione immagine

In tutti gli esperimenti svolti, si fa uso del metodo *createImage* (figura 6) che genera un immagine di dimensione width $\times$ height riempiendola con valori crescenti.

```
long long int* createImage(int height, int width) {
    long long int* image = (long long int*)malloc(width * height *
                                sizeof(long long int));
    for (int i = 0; i < width * height; i++) {
        image[i] = i + 1;
    }
    return image;
}
```

Figure 6. Metodo per la creazione dell'immagine

2.2. Scan sequenziale

In figura 7 è riportato il metodo *scan* per la versione sequenziale. Il metodo prende in ingresso il puntatore all'immagine, e le sue dimensioni e restituisce il risultato di scan sull'immagine. Nel

metodo sono usati due cicli for in modo da scorrere la matrice e accumulare i valori delle righe.

```
long long int *scan(long long int *image, int width, int height) {
    long long int tmp;

    for (int j = 0; j < height; j++) {
        tmp = 0;
        for (int i = 0; i < width; i++) {
            tmp += image[j * width + i];
            image[j * width + i] = tmp;
        }
    }

    return image;
}
```

Figure 7. Metodo di scan sequenziale

2.3. Transpose sequenziale

In figura 8 è riportato il metodo *transpose* per la versione sequenziale. Il metodo prende in ingresso il puntatore all'immagine e le sue dimensioni e restituisce la trasposta. Anche in questo caso due cicli for scorrono la matrice, e si fa uso di una matrice tmp di appoggio per invertire le righe e le colonne dell'input.

```
long long int *transpose(long long int *image, int width, int height) {
    long long int *tmp = (long long int *) malloc(width * height * sizeof(
    long long int));

    for (int j = 0; j < height; j++) {
        for (int i = 0; i < width; i++) {
            int index = i * height + j;
            tmp[index] = image[j * width + i];
        }
    }
    return tmp;
}
```

Figure 8. Metodo di transpose sequenziale

2.4. Versione sequenziale (scan-transpose)

Nella figure 9 è riportata una porzione del main per mostrare come vengano utilizzati i metodi scan e transpose per ottenere l'immagine integrale. Il metodo *imagePrint* si occupa di stampare i risultati delle varie operazioni.

```
image = scan(image, width, height);
imagePrint(image, width, height);

image = transpose(image, width, height);
imagePrint(image, width, height);

image = scan(image, width, height);
imagePrint(image, width, height);

image = transpose(image, width, height);
imagePrint(image, width, height);
```

Figure 9. Porzione di main della versione sequenziale

2.5. Scan parallelo

Per la versione parallela di *scan* è stato fatto riferimento all'implementazione fornita nelle slide del corso, integrando alcuni concetti presi dall'articolo di Nvidia [2].

```
void scan(long long int *d_input, long long int *d_output, long long int
        *sum,int inputSize, int blockSize){

    const int numBlocks = (inputSize + blockSize - 1) / blockSize;

    for (int i = 0; i < numBlocks; i++) {
        // Calcola l'offset in base al blocco corrente
        int offset = i * blockSize;

        // Esegui la scan sulla porzione corrente del vettore di input
        scanParallel<<<inputSize, blockSize>>>(d_input + offset, d_output +
                                               offset, inputSize, sum);
        // Esegui l'add sulla porzione corrente del vettore di output
        if (i < numBlocks - 1)
            add<<<inputSize, 1>>>(d_input + blockSize * (i + 1),
          inputSize, sum);
    }
}
```

Figure 10. Metodo scan per versione parallela

Questa versione sfrutta i thread della GPU per sommare gli elementi delle righe della matrice. Ogni riga viene processata da un numero di thread specificato in *blockSize*. Come discusso nell'articolo [2], nel caso in cui il numero di elementi da sommare in ogni riga sia superiore al numero di thread specificato in *blockSize*, è necessario introdurre una funzione *add()*.

Questa funzione si occupa di sommare il valore dell'ultimo elemento del blocco precedente al primo valore del blocco successivo. Questo è possibile perché l'ultimo elemento di una riga, calcolato da un blocco, contiene la somma parziale di tutti gli elementi precedenti di quella riga. Sommando questo valore parziale al blocco successivo, i nuovi elementi elaborati conterranno automaticamente la somma cumulativa dei valori precedenti.

In figura 11 è riportato il codice della versione parallelizzata CUDA di scan. Per eseguire in modo parallelo questo metodo, è necessario dividerlo in 2 fasi, una chiamata Up Sweep e l'altra Down Sweep.

Inoltre, l'array *sum* si occupa di registrare gli ultimi elementi di ogni riga, nel caso la riga non potesse essere processate in una volta sola.

2.5.1 Up Sweep

Durante la fase di Up Sweep, ciascun thread somma i suoi valori con i valori degli indici a una distanza di potenze di 2. Ad esempio, il thread con indice 0 somma il suo valore con quello del thread con indice 1, il thread con indice 2 somma il suo valore con quello del thread con indice 3, e così via. Questo processo viene eseguito

```
__global__ void scanParallel(long long int *d_input, long long int *
    d_output, int inputSize, long long int *sum) {
    __shared__ long long int tmp[2 * BLOCK_SIZE_S];
    int row = blockIdx.x;
    int idx = threadIdx.x;
    int offset = row * inputSize;

    tmp[2 * idx] = d_input[offset + idx * 2];
    tmp[2 * idx + 1] = d_input[offset + idx * 2 + 1];

    // Up Sweep
    for (unsigned int stride = 1; stride <= BLOCK_SIZE_S; stride *= 2) {
        __syncthreads();
        int index = (idx + 1) * stride * 2 - 1;
        if (index < 2 * BLOCK_SIZE_S)
            tmp[index] += tmp[index - stride];
    }

    // Down Sweep
    for (unsigned int stride = BLOCK_SIZE_S / 2; stride > 0; stride /= 2) {
        __syncthreads();
        int index = (idx + 1) * stride * 2 - 1;
        if (index + stride < 2 * BLOCK_SIZE_S) {
            tmp[index + stride] += tmp[index];
        }
    }
    __syncthreads();

    d_output[offset + idx] = tmp[idx];
    if (idx == BLOCK_SIZE_S - 1) {
        sum[row] = tmp[idx];
    }

}
```

Figure 11. Codice parallelizzato del metodo scan

in modo iterativo, raddoppiando la distanza tra i thread *(stride)* ad ogni iterazione, fino a quando la distanza non supera la dimensione della memoria condivisa. Durante questa fase, le somme parziali vengono inserite nell'array *tmp*.

2.5.2 Down Sweep

Nella fase di Down Sweep, l'array *tmp* ottenuto dalla fase di Up Sweep viene elaborato al contrario. Ad ogni iterazione, il valore del thread con indice corrente viene sommato con il valore del thread che si trova a una distanza di metà dello stride, dimezzando la distanza a ogni iterazione. L'effetto di questa operazione è la propagazione delle somme parziali agli elementi non ancora corretti. Come risultato si ha che nell'array *tmp* è contenuto il risultato finale di scan.

2.5.3 Add

Questa implementazione di scan rimane vincolata dal fatto che in CUDA il massimo numero di thread in esecuzione in un blocco è 1024. Per ovviare a questo problema, come analizzato nell'articolo [2], si può usare un array *sum*. In questa soluzione viene eseguito lo scan sulla prima porzione della riga, in modo che possa essere processata da un unico blocco. Una volta terminato il calcolo, viene salvato nell'array *sum* l'ultimo valore calcolato fino a quel momento (vedi riga 30 figura 11). Successivamente si usa il metodo *add* che somma l'elemento in *sum* al primo elemento del blocco successivo. In figura 12 è riportato il codice del metodo add.

```
__global__ void add(long long int *output, int length, long long int *n) {
    int blockID = blockIdx.x;

    int blockOffset = blockID * length;
    output[blockOffset] += n[blockID];
}
```

Figure 12. Metodo add versione parallela

2.6. Transpose Parallelo

Per la versione parallela di transpose, è stato fatto riferimento all'articolo [3], che suggerisce varie accortezza per migliorare le prestazioni di questo algoritmo. In figura 13 si vede come funziona transpose parallelizzato, dove la matrice viene divisa in blocchi (tile), e per ogni blocco viene calcolata la trasposta. La trasposta di ogni blocco viene poi collocata nella posizione corretta, come si vede in figura.

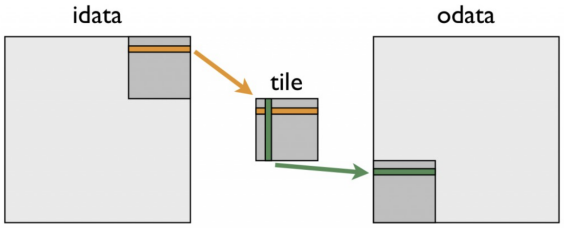

Figure 13. Meccanismo di transpose parallelizzato

In figura 14 è riportato il codice CUDA del metodo *transpose()*, e si può subito vedere un accortezza nell'allocazione della shared memory. Come si legge dall'articolo [3] infatti, questa soluzione è soggetta a conflitti nella shared memory, e che questo può essere risolto estendendo la dimensione della memoria condivisa aggiungendo una colonna.

In figura 15 è riportato un esempio dell'utilizzo del metodo *transpose*, in particolare delle definizioni della griglia e dei blocchi di thread. Come si può vedere, i blocchi sono quadrati di dimensione *TILE_SIZE_T* x *TILE_SIZE_T*, mentre la griglia viene calcolata dividendo *INPUT_SIZE / TILE_SIZE_T* in modo da ottenere il numero di tile nella griglia.

```
__global__ void transpose(long long int *idata,long long int *odata) {
    __shared__ long long int tile[TILE_SIZE_T][TILE_SIZE_T + 1]; // con
    TILE_SIZE + 1 si rimuovono i bank conflicts

    int x = blockIdx.x * TILE_SIZE_T + threadIdx.x;
    int y = blockIdx.y * TILE_SIZE_T + threadIdx.y;
    int width = gridDim.x * TILE_SIZE_T;

    for (int j = 0; j < TILE_SIZE_T; j += BLOCK_SIZE_T)
        tile[threadIdx.y + j][threadIdx.x] = idata[(y + j) * width + x];

    __syncthreads();

    x = blockIdx.y * TILE_SIZE_T + threadIdx.x;
    y = blockIdx.x * TILE_SIZE_T + threadIdx.y;

    for (int j = 0; j < TILE_SIZE_T; j += BLOCK_SIZE_T) {
        odata[(y + j) * width + x] = tile[threadIdx.x][threadIdx.y + j];
    }
}
```

Figure 14. Metodo transpose versione parallela

```
dim3 dimGrid(INPUT_SIZE / TILE_SIZE_T, INPUT_SIZE / TILE_SIZE_T, 1);
dim3 dimBlock(TILE_SIZE_T, BLOCK_SIZE_T, 1);

transpose<<<dimGrid, dimBlock>>>(d_outputS, d_outputT1);
```

Figure 15. Uso di transpose

2.7. Versione sequenziale (ottimizzata)

In figura 16 si può vedere il codice che realizza la versione sequenziale ottimizzata di integral image. La riga 15 mostra come l'elemento in posizione (i,j) venga calcolato a partire dai risultati parziali contenuti nelle celle (i-1,j), (i,j-1) e (i-1,j-1).

```
long long int *integralImage_optimized(long long int *image, long long int
    *integImg, int width, int height) {

    // Calcolo la prima riga
    integImg[0] = image[0];
    for (int i = 1; i < width; i++)
        integImg[i] = integImg[i - 1] + image[i];

    // Calcolo la prima colonna
    for (int i = 1; i < height; i++)
        integImg[i * width] = integImg[(i - 1) * width] + image[i * width];

    // Calcolo gli elementi restanti
    for (int i = 1; i < height; i++) {
        for (int j = 1; j < width; j++) {
            integImg[i * width + j] = (integImg[(i - 1) * width + j] +
                                       integImg[i * width + j - 1] -
                                       integImg[(i - 1) * width + j - 1] +
                                       image[i * width + j]);
        }
    }

    return integImg;
}
```

Figure 16. Versione sequenziale ottimizzata

3. Sperimentazione

Gli esperimenti condotti per confrontare il codice sequenziale e quello parallelizzato tramite CUDA sono di due tipi:

- Variazione della configurazione dei thread in blockDim: in questi esperimenti si stabilisce una dimensione fissa per l'immagine, per poi eseguire la versione CUDA di integral image variando le dimensioni dei blocchi di thread;
- Variazione della dimensione dell'immagine: in questi esperimenti viene scelta una configurazione di thread ed eseguito lo script sequenziale e parallelo su immagini sempre più grandi.

Questi esperimenti sono stati condotti sia nel confronto fra versioni scan-transpose che nel confronto con la versione sequenziale ottimizzata. Tutti gli esperimenti sono stati eseguiti con una CPU Intel® Core™ i7-8565U e una GPU NVIDIA RTX A2000 12GB che disponeva di 3328 cores.

3.1. Scan-Transpose sequenziale vs Parallela

3.1.1 Variare la configurazione dei thread

Questo esperimento è stato condotto usando un'immagine di dimensione 2048x2048. La versione sequenziale ha impiegato 235ms. In figura 17 è riportato l'andamento del tempo di esecuzione della versione parallela. Sull'asse delle ascisse sono riportate le configurazioni dei threadBlock per gli algoritmi di scan e transpose. Ad esempio il valore (32,32,128) indica che sono stati usati blocchi di dimensione 32x32 per il calcolo di transpose e blocchi 128x1 per il calcolo di scan. Passando ad osservare l'andamento del grafico, si vede chiaramente che, aumentando i thread in esecuzione, il tempo impiegato dall'algoritmo diminuisce rapidamente. In figura 18 si può osservare il grafico dello speed up relativo a questo esperimento. L'andamento dello speed up tende a crescere all'aumentare del numero di thread, quasi fino al valore 100. É interessante osservare che in corrispondenza del valore massimo (32,32,1024) lo speed up cala a 80. Questo fenomeno può essere spiegato dal fatto che, aumentando eccessivamente il numero di thread, si ha un maggiore overhead di sincronizzazione che porta a maggiori tempi di attesa.

3.1.2 Variare il numero di pixel

Per questo esperimento la configurazione dei thread scelta è stata 32x32 thread per il calcolo di transpose e 256 thread per il calcolo di scan. Gli esperimenti sono stati condotti partendo da immagini 128x128 e raddoppiando le immagini

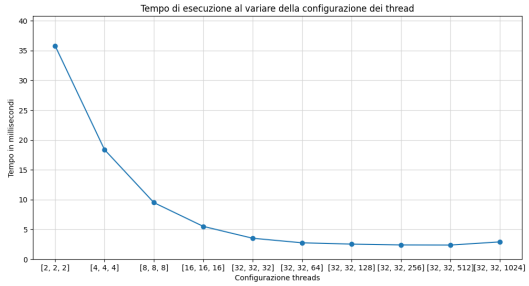

Figure 17. Tempo di esecuzione della versione parallela al variare della configurazione dei thread

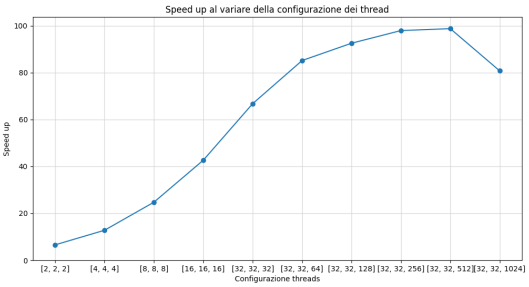

Figure 18. Speed up della versione parallela al variare della configurazione dei thread

fino a 8192x8192. Come si può vedere dalle figure 19 e 20, l'andamento dei tempi di esecuzione è lineare, ma la versione sequenziale arriva ad impiegare più di 2500ms mentre quella parallela si ferma intorno ai 35ms. Dalla figura 21 si osserva l'andamento dello speed up al variare del numero di pixel. In questa configurazione, si raggiunge il massimo speed up in corrsipondenza dell'immagine 2048x2048. Una volta superata questa soglia il grafico tende a scendere, come visto anche nell'esperimento precedente.

3.2. Sequenziale ottimizzata vs Scan-Transpose Parallela

3.2.1 Variare la configurazione dei thread

Anche per questo confronto è stato condotto l'esperimento sull'immagine di dimensione 2048x2048 variando la configurazione dei thread. In questo caso la versione ottimizzata sequenziale ha impiegato 36ms. Da questo risultato e dai tempi impiegati dalla versione parallela (figura 16) è stato calcolato il grafico in figura 22.

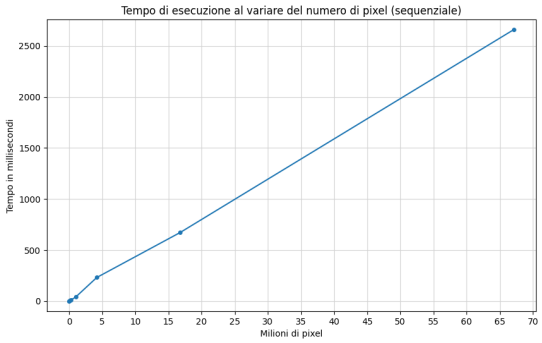

Figure 19. Tempo di esecuzione della versione sequenziale al variare del numero di pixel

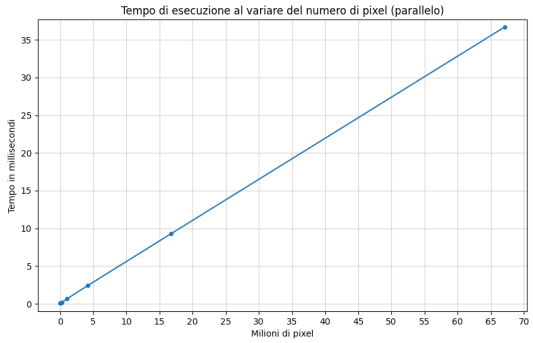

Figure 20. Tempo di esecuzione della versione parallela al variare del numero di pixel

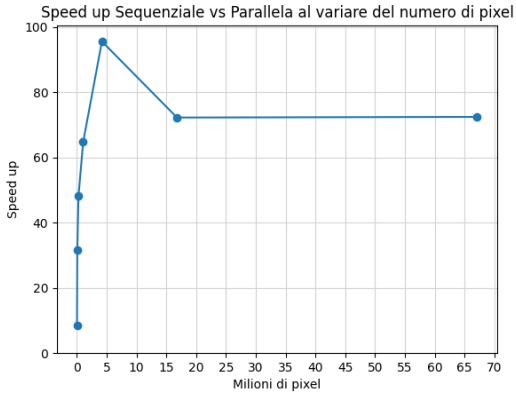

Figure 21. Speed up al variare del numero di pixel

La figura mostra lo speed up della versione parallela rispetto a quella sequenziale ottimizzata. L'andamento è lo stesso rispetto al confronto fra scan-transpose, visto che sono stati usati gli stessi tempi per la versione parallela. Da questo confronto emerge che per la configu-

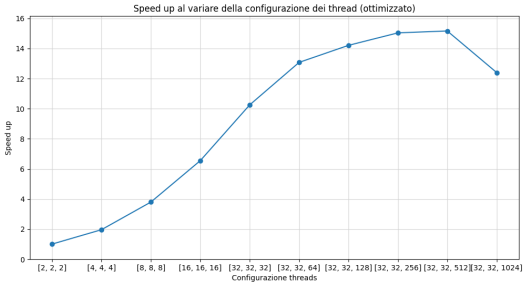

Figure 22. Speed up al variare della configurazione dei thread con la versione sequenziale ottimizzata.

razione (32,32,512) si raggiunge uno speed up di 15, un risultato notevole considerando che la versione ottimizzata accede direttamente ai risultati parziali calcolati in precedenza.

3.2.2 Variare il numero di pixel

Per questo esperimento, la configurazione dei thread è stata nuovamente 32x32 thread per il calcolo di transpose e 256 thread per il calcolo di scan. In figura 23, si ha il confronto fra i tempi di esecuzione della versione parallela e di quella sequenziale ottimizzata. Si può osservare chiaramente che la curva della versione parallela sia molto più bassa rispetto a quella della versione sequenziale. In figura 24 si osserva l'andamento dello speed up relativo a questo esperimento. Anche in questo caso si ha un massimo in corrispondenza della dimensione 2048x2048, dopo il quale lo speed up tende a scendere, come per gli esperimenti precedenti. Il valore del massimo è 15, come quello ottenuto nell'esperimento precedente.

4. Conclusioni

Al termine di questa analisi si può concludere che CUDA fornisce un notevole vantaggio nella risoluzione del problema del calcolo dell'immagine integrale. Difatti, per immagini con risoluzione alta, si arriva a uno speed up di circa 15. É possibile anche concludere che, per ottenere le migliori prestazioni, è importante effettuare vari esperimenti per trovare la configurazione migliore per i thread CUDA.

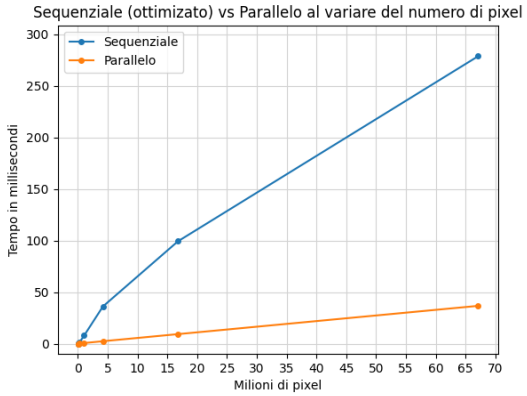

Figure 23. Confronto dei tempi di esecuzione al variare del numero di pixel.

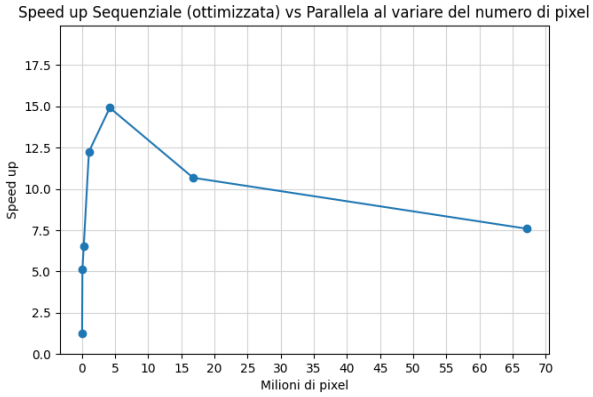

Figure 24. Speed up al variare del numero di pixel con la versione sequenziale ottimizzata.

References

[1] Berkin Bilgic, Berthold K.P. Horn, Ichiro Masaki *Efficient Integral Image Computation on the GPU*. https://nmr.mgh.harvard.edu/~berkin/Bilgic_2010_Integral_Image.pdf

[2] Mark Harris, Shubhabrata Sengupta, John D. Owens *Parallel Prefix Sum (Scan) with CUDA*. https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda

[3] Mark Harris *An Efficient Matrix Transpose in CUDA C/C++*. https://developer.nvidia.com/blog/efficient-matrix-transpose-cuda-cc