Integral Image Generator

Parallel Programming for Machine Learning
Project Work in Artificial Intelligence Programming

Sofia Galante
sofia.galante@stud.unifi.it

1 Introduzione

Il progetto svolto è un generatore di immagini integrali.
In un'immagine integrale, data un'immagine di partenza, ogni pixel è ottenuto sommando se stesso con tutti i pixel precedenti (sia lungo l'asse x che lungo l'asse y).
Si sono creati diversi generatori di immagini integrali:

1. *un generatore sequenziale*: in questo caso si generano le immagini integrali con un algoritmo sequenziale;

2. *due versioni di un generatore con CUDA*: in questo caso si utilizza la GPU per parallelizzare la creazione dell'immagine integrale;

3. *tre versioni di un generatore con OpenMP*: in questo caso si compie una parallelizzazione a livello della CPU del codice tramite OpenMP.

Lo scopo del progetto è quello di osservare lo speedup ottenuto nelle due versioni parallele dell'algoritmo, osservando anche quale parallelizzazione (tra GPU e CPU) risulta più efficiente.
Tutti gli esperimenti sono stati svolti su un PC con sistema operativo Windows 10, una CPU Intel Core i5-11400 e una GPU RTX 3070Ti.
La versione sequenziale e quella parallelizzata a livello di GPU utilizzano Visual C++ come compilatore, in quanto CUDA è compatibile solo con questo.
Per quanto riguarda la versione con OpenMP, si è optato per l'utilizzo di MinGW, in quanto Visual C++ dava problemi nell'annidare i cicli di parallel for.

2 Organizzazione del codice

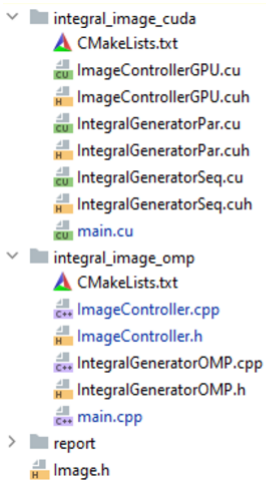

Figura 1: Organizzazione del codice

Il codice è stato suddiviso in due diversi progetti (si è utilizzato CLion come IDE): un progetto che utilizza Visual C++, e uno che utilizza MinGW. Inoltre un file .h è condiviso da entrambi i progetti.

2.1 Il file condiviso: Image.h

Il file condiviso da entrambi i progetti è il file **Image.h**, il quale contiene al suo interno la definizione di immagine, cioè una struct che ha come attributi una larghezza (*width*), un'altezza (*height*) e un puntatore di interi (*pixels*) ad una locazione in memoria, la quale viene allocata dinamicamente durante l'esecuzione del programma.

Si noti che l'allocazione del puntatore di interi viene svolta con *malloc()* se l'immagine è salvata nella CPU e con *cudaMalloc()* se è invece salvata sulla GPU.

Questo file contiene anche una variabile globale (*SEED*), utilizzata come seed per la generazione di numeri casuali in entrambi i progetti, così da generare le stesse immagini.

2.2 I file del progetto con compiler Visual C++

Il primo progetto svolge i test di tipo sequenziale e quelli di tipo parallelo su GPU e li salva in appositi file csv.
I file presenti in questo progetto sono:

1. **ImageController.cuh/cu**: i file che contengono la definizione e la dichiarazione di tutte le funzioni che si possono svolgere su un'Image. In particolare, si hanno una funzione per generare un'immagine casuale (*generateImage()*), una funzione per copiare un'immagine (*copyImage()*), funzioni per allocare/deallocare un'immagine dalla memoria (*allocateImage(), freeHostImage(), freeDeviceImage()*) e funzioni per copiare un'immagine sulla CPU dalla GPU e viceversa (*copyFromDeviceToHost() e copyFromHostToDevice()*.
2. **IntegralGeneratorSeq.cuh/cu**: i file che contengono la definizione e la dichiarazione della funzione necessaria a svolgere l'algoritmo sequenziale (*generateIntegralCPUseq()*);
3. **IntegralGeneratorPar.cuh/cu**: i file che contengono la definizione e la dichiarazione di due funzioni che implementano due diversi algoritmi paralleli con CUDA (*generateIntegralGPUglobalMem() e generateIntegralGPUsharedMem()*) e le funzioni *setup()* e *finish()* utilizzate per allocare, copiare e deallocare tutte le immagini necessarie su CPU e GPU; come detto sopra, si sono scritte due diverse implementazioni dell'algoritmo di tipo parallelo e, di questo, ne parleremo nel dettaglio nella sezione dedicata (la 3);
4. **main.cu**: il file contenente tutte le funzioni relative ai test da svolgere e la funzione *main()* del programma; si noti che quest'ultima riceve dall'esterno il nome del file con cui salvare i test svolti.

2.3 I file del progetto con compiler MinGW

Il secondo progetto si occupa di svolgere i test di tipo parallelo su CPU (con l'utilizzo di OpenMP) e di scrivere il risultato in relativi file csv.
I file presenti in questo progetto sono:

1. **ImageController.h/cpp**: questi file contengono le funzioni riguardanti le immagini salvate su CPU presenti in ImageController.cuh del primo progetto.
2. **IntegralGeneratorOMP.h/cpp**: i file che contengono al loro interno la dichiarazione e la definizione di tre funzione che svolgono l'algoritmo parallelo con OpenMP (*generateIntegralCPUompFirstLevel(), generateIntegralCPUompSecondLevel() e generateIntegralCPUompBothLevels()*).
3. **main.cpp**: il file contenente tutte le funzioni relative ai test da svolgere e la funzione *main()* del programma; si noti che quest'ultima riceve dall'esterno il nome del file con cui salvare i test svolti.

3 Parallelizzazione con CUDA

In **IntegralGeneratorPar** si trovano due diverse implementazioni dello stesso algoritmo parallelo.

La prima di queste (*generateIntegralGPUglobalMem()*) è una parallelizzazione diretta della versione sequenziale dell'algoritmo: in questo caso ogni thread processa un singolo elemento dell'immagine e l'immagine originale e quella risultante si trovano entrambe nella memoria globale.
Questo tipo di algoritmo è penalizzato dai numerosi accessi alla **global memory** (uno ogni volta che si legge un elemento dell'immagine) che causano un aumento dell'effettivo tempo necessario.

```cpp
__global__ void generateIntegralGPUglobalMem(int width, int height,
                                             int const * original, int * result){
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    int row = by * blockDim.y + ty;
    int col = bx * blockDim.x + tx;

    int value = 0;
    if(col < width && row < height){
        for(int y = row; y >= 0; y--){
            for(int x = col; x >= 0; x--){
                value += original[y * width + x];
            }
        }
        result[row * width + col] = value;
    }
}
```

Figura 2: CUDA - Global Memory

La seconda versione dell'algoritmo (*generateIntegralGPUsharedMem()*) utilizza la **shared memory** e la tecnica del **tiling** per risolvere il rallentamento dato dai continui accessi in global memory. La shared memory utilizzata viene allocata dinamicamente e dipende dalle dimensioni del blocco.
In questo algorimto, i thread di un blocco collaborano per caricare in shared memory gli elementi dell'immagine originale necessari per il calcolo. In particolare, l'algoritmo è stato diviso in fasi: ogni fase riguarda il caricamento di una quantità di elementi in shared memory pari alla grandezza di un block. Il numero di fasi di ogni block dipende dalla sua posizione nella grid: esso deve infatti caricare un numero di blocchi in memoria pari alla cardinalità dell'insieme di blocchi composto da sé stesso e dai blocchi che lo precedono (cioè che hanno indice x e/o y inferiore al suo).

```cpp
__global__ void generateIntegralGPUsharedMem(int width, int height, int const * original,
                                             int * result){
    //Utilizzo della shared memory
    int bx = blockIdx.x;
    int by = blockIdx.y;
    int tx = threadIdx.x;
    int ty = threadIdx.y;
    int w = blockDim.x;
    int h = blockDim.y;

    int row = by * h + ty;
    int col = bx * w + tx;
    extern __shared__ int sharedOriginal[];
    int value = 0;
    int _row, _col, _x, _y;
    for(int _by = 0; _by <= by; _by++){
        for(int _bx = 0; _bx <= bx; _bx++){ //itera tra i blocchi
            _row = _by * h + ty;
            _col = _bx * w + tx;
            if(_col < width && _row < height)
                sharedOriginal[ty * w + tx] = original[_row * width + _col];
            else
                sharedOriginal[ty * w + tx] = 0;
            __syncthreads();//ogni thread scrive nella shared memory e poi aspetta
```

Figura 3: CUDA - Shared Memory (Prima parte)

Una volta caricati i blocchi in shared memory, i vari thread calcolano un valore parziale del risultato e, alla fine di tutte le fasi, ogni thread copia il valore calcolato nel pixel dell'immagine risultante.

```cpp
            if(_bx < bx){ _x = w-1;}
            else{ _x = tx;}

            if(_by < by){_y = h-1;}
            else{_y = ty;}
            if(col < width && row < height){
                for(int y = _y; y >= 0; y--){
                    for(int x = _x; x >= 0; x--){
                        value += sharedOriginal[y * w + x];
                    }
                }
            }
            __syncthreads();
        }
    }
    if(col < width && row < height)
        result[row * width + col] = value;
}
```

Figura 4: CUDA - Shared Memory (Seconda parte)

Si noti che, a differenza dell'algoritmo precedente, in questo sono presenti due chiamate alla funzione di sincronizzazione dei thread: questa funzione è necessaria in quanto, se non ci fosse, alcuni thread potrebbero (1) iniziare a calcolare il valore parziale del risultato prima che la shared memory sia stata tutta copiata e/o (2) iniziare a riscrivere la shared memory prima che un altro thread abbia finito il suo calcolo. Inserendo quelle due sincronizzazioni, questi fatti vengono evitati.

Infine si vuole far notare che per osservare un effettivo miglioramento nell'utilizzo del secondo algoritmo rispetto al primo si è dovuta disattivare manualmente attraverso il CMake la cache di primo livello: il compilatore infatti salvava nella cache gli elementi da leggere, rendendo il primo algoritmo veloce quanto (se non, in alcuni casi, più) del secondo.

```
void generateIntegralCPUompBothLevels(int width, int height, int const * original,
                                      int * result, int threads1, int threads2){
#pragma omp parallel for collapse(2) default(none) shared(original, result) \
firstprivate(width, height, threads2) num_threads(threads1)
    for(int row = 0; row < height; row++){
        for(int col = 0; col < width; col++){
            int value = 0;
#pragma omp parallel for collapse(2) default(none) shared(original) \
firstprivate(row, col, width) num_threads(threads2) reduction(+: value)
            for(int y = row; y >= 0; y--){
                for(int x = col; x >= 0; x--){
                    value += original[y * width + x];
                }
            }
            result[row * width + col] = value;
        }
    }
}
```

Figura 5: OpenMP - parallelizzazione a entrambi i livelli

4 Parallelizzazione con OpenMP

La parallelizzazione dell'algoritmo svolta con l'utilizzo di OpenMP si trova in **IntegralGeneratorOMP**.
In questo caso, la parallelizzazione è stata effettuata con due diverse direttive *omp parallel for*, annidate una dentro l'altra.

La prima direttiva *omp parallel for* (annidata a due livelli) parallelizza i due cicli for che danno le coordinate dell'elemento dell'immagine da calcolare.
Questa parallelizzazione è stata chiamata nel programma parallelizzazione al **primo livello**.

La seconda direttiva *omp parallel for* (anche questa annidata a due livelli) parallelizza invece il calcolo dell'elemento selezionato in precedenza. Visto che il calcolo di ogni pixel deriva da una somma di tutti i pixel precedenti, per sfruttare al meglio le risorse di OpenMP si è deciso di utilizzare una *reduction*.
Questa parallelizzazione è stata chiamata nel programma parallelizzazione al **secondo**.

Si noti che la scrittura nell'immagine risultante non è stata marcata come sezione critica in quanto ogni thread modifica un diverso elemento dell'immagine, non causando quindi alcuna race condition.

Per osservare tutti i possibili risultati, si è deciso di creare tre diversi algoritmi di generazione dell'immagine integrale:

1. *generateIntegralCPUompFirstLevel()*, che contiene solo la parallelizzazione del primo livello;
2. *generateIntegralCPUompSecondLevel()*, che contiene solo la parallelizzazione del secondo livello;
3. *generateIntegralCPUompBothLevels()*, che contiene entrambe le parallelizzazioni (ed è la funzione riportata in figura).

5 Esperimenti svolti

Per osservare lo *speedup* ottenuto dalla parallelizzazione del codice si sono compiuti diversi esperimenti.
In particolare si sono svolti tre diversi test, ognuno dei quali studia l'aumento del tempo di esecuzione al variare di alcuni parametri.
I test svolti sono:

1. un test che osserva come il tempo di esecuzione per i due algoritmi su GPU si modifichi al variare della dimensione dei block e della grid e quale dei due algoritmi sia il migliore;
2. un test che confronta i diversi livelli di parallelizzazione dell'algoritmo di OpenMP, trovando il migliore e facendo poi un test su quest'ultimo per vedere come si comporta all'aumentare dei thread;

3. un test che confronta il tempo di esecuzione dell'algoritmo sequenziale, del miglior algoritmo CUDA e del miglior algoritmo OpenMP al variare della dimensione del labirinto.

Tutti gli esperimenti sono stati lanciati dal programma python **tests.py**. Ogni esperimento è stato svolto 10 volte. I risultati sono stati ottenuti facendo la media del tempo di esecuzione di ogni run (dopo aver eliminato i tempi massimi e minimi ottenuti). Tutti i risultati (sia quelli delle singole iterazioni dei test, sia quelli finali) sono salvati in appositi file csv. Per gli esperimenti si sono utilizzate delle immagini quadrate.
Il programma ha anche generato i grafici riportati in seguito.

5.1 Variazione delle dimensioni dei block e della grid - CUDA

Questo esperimento studia come si modificano i tempi di esecuzione dei due algoritmi CUDA al variare della dimensione della grid e dei block.
Si sono svolte due diverse versioni di questo esperimento.

5.1.1 Studio dei casi limite

Nella prima versione si sono scelte tre immagini di dimensione fissa (10x10, 100x100 e 1000x1000) e si sono osservati i casi limite per le dimensioni di block e grid, in particolare si sono studiati i seguenti casi:

- **immagine divisa in righe**: in questo caso l'immagine è stata divisa in righe e ogni riga assegnata a un blocco;

- **immagine divisa in colonne**: in questo caso l'immagine è stata divisa in colonne e ogni colonna è stata assegnata a un blocco;

- **massimizzazione del numero di block nella grid**: in questo caso si sono usati più block possibili;

- **massimizzazione del numero di thread per block**: in questo caso si sono usati più thread per block possibili.

I valori effettivi delle dimensioni di grid e block per le tre immagini possono essere visti nella tabella qui sotto:

		dim 10x10	dim 100x100	dim 1000x1000
righe	grid	1x10	1x100	1x1000
	block	10x1	100x1	1000x1
colonne	grid	10x1	100x1	1000x1
	block	1x10	1x100	1x1000
max n° blocks	grid	10x10	100x100	255x255
	block	1x1	1x1	4x4
max n° threads	grid	1x1	4x4	32x32
	block	10x10	32x32	32x32

Tabella 1: Tabella con dimensioni di griglia e blocco per ogni immagine

Si noti che in alcuni casi nella massimizzazione del numero di block e nella massimizzazione del numero di thread i valori non corrispondono alla grandezza effettiva dell'immagine: questo accade in quanto, se si usassero la larghezza e l'altezza dell'immagine, si supererebbero i valori limite per numero di thread in un block (1024) e/o per numero di block in una grid (65535).

I risultati ottenuti sono i seguenti:

	GPU - global	GPU - shared
1 x 10 - 10 x 1	0.047 ms	0.039 ms
10 x 1 - 1 x 10	0.041 ms	0.046 ms
10 x 10 - 1 x 1	0.056 ms	0.061 ms
1 x 1 - 10 x 10	0.042 ms	0.038 ms

Tabella 2: Immagine 10 x 10

	GPU - global	GPU - shared
1 x 100 - 100 x 1	0.236 ms	0.131 ms
100 x 1 - 1 x 100	1.005 ms	0.808 ms
100 x 100 - 1 x 1	1.054 ms	11.78 ms
4 x 4 - 32 x 32	0.715 ms	0.386 ms

Tabella 3: Immagine 100 x 100

	GPU - global	GPU - shared
1 x 1000 - 1000 x 1	829.907 ms	202.801 ms
1000 x 1 - 1 x 1000	5500.471 ms	912.196 ms
255 x 255 - 4 x 4	1783.244 ms	969.084 ms
32 x 32 - 32 x 32	807.314 ms	229.303 ms

Tabella 4: Immagine 1000 x 1000

Osservando le tre tabelle ottenute, si possono fare le seguenti considerazioni:

1. L'algoritmo con memoria di tipo shared è più veloce di quello con memoria globale, tranne quanto la dimensione dei blocchi è troppo piccola (nei due casi in cui i blocchi hanno dimensioni 1 x 1, questo algoritmo è più lento); questo può essere spiegato facilmente dal fatto che, se il blocco è di dimensioni unitarie, allora la shared memory è inutile da utilizzare, in quanto si deve accedere ai valori copiati solo una volta. Si noti che basta che la dimensioni aumenti di poco (come nel caso 4 x 4) per far tornare vantaggioso l'utilizzo della shared memory.

2. In quasi tutti i casi, la divisione in righe comporta algoritmi più veloci della divisione in colonne; ciò è dato dal fatto che, nella divisione in righe, thread consecutivi devono accedere in posizioni consecutive della memoria (fanno degli accessi *coalesced*) permettendo alle DRAM di rendere disponibili i dati con una burst (il tempo per accedere in memoria viene ridotto a quello di un singolo accesso). Nel caso della divisione per colonne, invece, questo non accade e i tempi si allungano.

3. Massimizzare la grandezza dei blocchi sembra essere molto più efficiente (all'aumentare dell'immagine) di massimizzare la grandezza della griglia.

5.1.2 Studio di block di dimensioni multiple di 32

Nella seconda versione del test ci siamo concentrati su dimensioni quadrate multiple di 32 per i block.
Questa scelta è stata dettata dal fatto che i thread all'interno di un block sono divisi a gruppi di 32 nei warp. Utilizzare una dimensione di block diversa da un multiplo di 32, quindi, comporterebbe comunque alla generazione di un numero di thread multipli di 32 per quel block, i quali resterebbero inattivi durante il processo.
Per osservare quale tra le tre dimensioni quadrate ammissibili multiple di 32 (8x8, 16x16 e 32x32) fosse la migliore per il nostro esperimento, si sono svolte più iterazioni dei due algoritmi CUDA (una per ognuna delle tre dimensioni) e si è osservato come i tempi di esecuzione variassero all'aumentare della dimensione dell'immagine (la quale parte da 10x10 e arriva fino a 500x500, aumentando il lato di 10 in 10).

Il grafico totale ottenuto è il seguente:

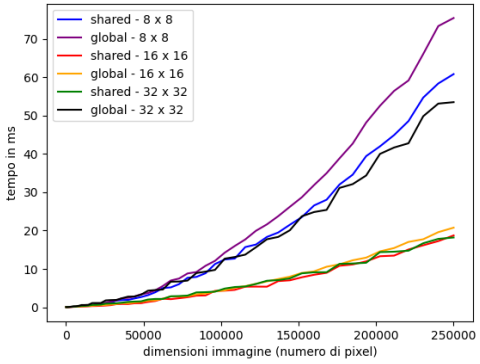

Figura 6: Tempo impiegato dall'esecuzione parallela al variare della dimensione dell'immagine.

da cui si può osservare come gli algoritmi con shared memory siano superiori, a parità di grandezza dei block, a quelli che utilizzano solo la global memory, come ci aspettavamo. In alcuni casi, questa superiorità si nota anche per dimensioni del block non uguali (l'algoritmo con la shared memory con dimensione dei block pari a 32x32 è migliore, per dimensioni elevate dell'immagine, sia del corrispettivo algoritmo con global memory, sia di quello con dimensione di block pari a 16x16).

Per studiare più nel dettaglio i risultati ottenuti, si è deciso di dividere il grafico in due: uno per l'algoritmo che utilizza la shared memory e uno per l'algoritmo che utilizza la global:

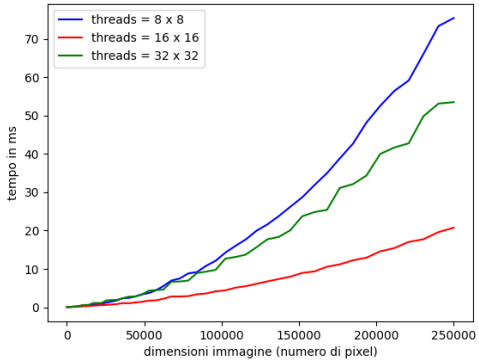

Figura 7: Tempo impiegato dall'esecuzione parallela (con memoria globale) al variare della dimensione dell'immagine.

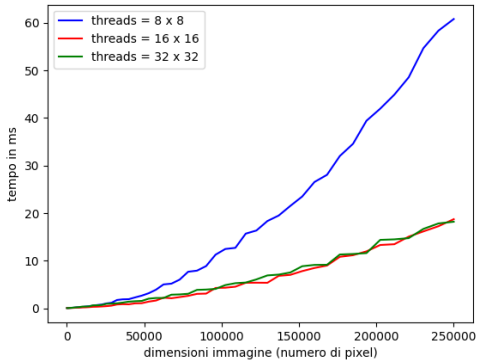

Figura 8: Tempo impiegato dall'esecuzione parallela (con memoria shared) al variare della dimensione dell'immagine.

Come si può osservare dai due grafici, in entrambi i casi la dimensione 8x8 si rivela essere la meno efficiente. Questo è dovuto al fatto che i block non sono occupati con una percentuale efficiente, comportando, quindi, una perdita di prestazioni dell'algoritmo.

Per quanto riguarda la dimensione massima invece (32x32), questa comporta un miglioramento evidente nella versione shared, mentre continua a non essere così veloce per la versione con la global memory.
Questo miglioramento nella versione shared memory è dato dal fatto che la memoria allocata, essendo maggiore, permette meno accessi (non consecutivi) alla memoria globale e diminuisce il numero di fasi necessarie alla conclusione dell'algoritmo, portando a un miglioramento nei tempi.

Infine, si può notare che la dimensione di mezzo (16x16) è la migliore per entrambi gli algoritmi.
Visto che tutti i thread nello stesso block eseguono le stesse azioni contemporaneamente, lo speedup ottenuto nella versione con global memory può essere spiegato dal fatto che il tempo impiegato per la gestione di 1024 thread da parte di un blocco è molto maggiore di quello impiegato per la gestione di soli 256 thread: il numero inferiore di block nel primo caso non porta ad una diminuzione del tempo di esecuzione abbastanza elevata da contrastare il tempo di gestione dei thread.
Per quanto riguarda la shared memory, invece, si può osservare che (nonostante la memoria allocata sia inferiore rispetto allo scorso caso), la velocità di esecuzione è simile a quella del caso precedente. La motivazione dietro a ciò riguarda, oltre alla gestione del numero dei thread da parte del block come per la global memory, anche il fatto che in questa versione dell'algoritmo siano presenti dei punti di sincronizzazione per tutti i thread di un block: in questo caso avere più block è vantaggioso, così mentre i thread di un block sono in attesa, quelli di un altro block possono continuare a operare sui dati.

Viste queste considerazioni, la dimensione dei block scelta per il test precedente (quello riguardante la variazione della dimensione dell'immagine) è stata la dimensione 16x16.

5.2 Variazione del numero di thread - OpenMP

Il secondo test svolto riguarda la modifica nel numero di thread nell'algoritmo parallelizzato tramite OpenMP.
Anche di questo test sono state fatte due versioni.

5.2.1 Studio del livello di parallelizzazione

Per capire quale dei tre possibili algoritmi OpenMP fosse il migliore, si è svolto un test in cui si osservano i tempi di esecuzione al variare del livello di parallelizzazione e del numero di thread utilizzati.

Si sono lanciati cinque diversi algoritmi, con i seguenti parametri:

1. primo livello - 15 thread;
2. secondo livello - 15 thread
3. entrambi i livelli - 15 thread nel primo livello, 1 nel secondo;
4. entrambi i livelli - 1 thread nel primo livello, 15 nel secondo;
5. entrambi i livelli - 15 thread nel primo livello, 15 nel secondo;

Ognuno di questi cinque algoritmi è stato fatto lavorare su immagini diverse per osservare come l'aumentare delle dimensioni (che vanno da 10x10 a 100x100 aumentando il lato di 5 in 5) influisse sui tempi di esecuzione.
Il grafico è in scala logaritmica lungo l'asse y
Di seguito il risultato:

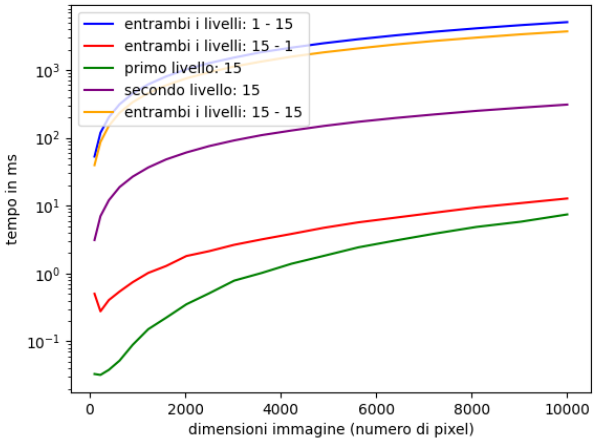

Figura 9: Studio del livello di parallelizzazione degli algoritmi OpenMP

Osservando il grafico ottenuto, si possono fare le seguenti considerazioni:

1. La parallelizzazione più efficiente è quella svolta al primo livello. Si noti che:
   (a) la parallelizzazione al primo livello è migliore di quella al secondo livello perché la seconda viene invocata più volte (il tempo di gestione dei thread influisce maggioramente) e inoltre la seconda compie una reduction (più pesante del normale parallel for).
   (b) la parallelizzazione al primo livello è migliore di quella a entrambi i livelli perché la seconda comporta un'esplosione del numero di thread, andando in overhead e aumentando molto i tempi di esecuzione.
2. Mentre i tempi di esecuzione dell'algoritmo con parallelizzazione al primo livello e quello con una parallelizzazione a entrambi livelli con un solo thread nel secondo livello è simile (il fatto che il secondo algoritmo abbia un tempo di esecuzione leggermente maggiore è dato, ovviamente, dal dover gestire due diverse parallel region), lo stesso non si può dire per l'algoritmo con parallelizzazione al secondo livello e quello con parallelizzazione a entrambi livelli ma con un solo thread al primo livello. L'enorme differenza dei tempi (controintuitiva), in questo caso, può essere data da un'ottimizzazione diversa del compilatore per la generazione dei thread al secondo livello nei due diversi casi.

Da queste considerazioni, si deduce che il modo migliore per parallelizzare l'algoritmo con OpenMP è quello al primo livello.

5.2.2 Studio del numero dei thread

In questa seconda versione dell'esperimento si è osservato quale numero di thread potesse essere migliore per l'algoritmo di parallelizzazione al primo livello.
In particolare si sono lanciati tre versioni dell'algoritmo (uno con 10 thread, uno con 100 e uno con 1000) e si sono osservati i loro tempi di esecuzione al variare della dimensione dell'immagine (da 10x10 a 500x500, aumentando il lato di 10 in 10).

Questi i risultati ottenuti:

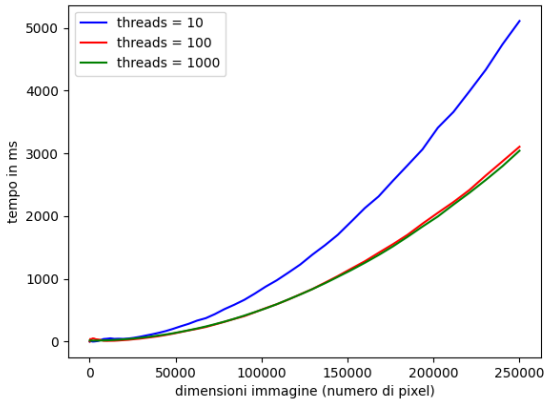

Figura 10: Studio del numero dei thread

Come si può notare, l'aumento del numero di thread (anche molto elevato) non causa un rallentamento del programma, ma, mentre i tempi di esecuzione con 10 thread sono nettamente maggiori, la differenza dei tempi di esecuzione per gli altri due casi è quasi inesistente.
Questo è dato dal fatto che più è elevato il numero di thread, maggiore sarà anche il tempo di gestione di essi, e lo speedup ottenuto tenderà a annullarsi (e, con l'aumentare dei thread, a diventare negativo).
Vista la poca differenza dell'algoritmo tra l'utilizzo di 100 e 1000 thread, si è deciso di utilizzare il primo numero di thread nell'esperimento seguente.

5.3 Variazione delle dimensioni dell'immagine

Infine, in questo esperimento si è osservato lo speedup ottenuto dalla parallelizzazione su GPU (con shared memory) e da quella su CPU (al primo livello) a confronto con l'algoritmo sequenziale all'aumentare della dimensione dell'immagine.

Il valore dei parametri di interesse è:

- dimensioni dell'immagine che partono da 10x10 e arrivano fino a 500x500, aumentando il lato di 10 in 10 (quindi si sono svolte 50 iterazioni);
- numero di thread per l'algoritmo in versione OpenMP = 100;
- dimensioni dei block per CUDA = 16x16 (la grandezza della grid viene calcolata di conseguenza in base all'immagine), il miglior risultato dei test sulla dimensione dei block per CUDA.

Per questo grafico, vista l'enorme differenza tra i tempi di esecuzione, si è deciso di utilizzare una scala logaritmica lungo l'asse delle y.

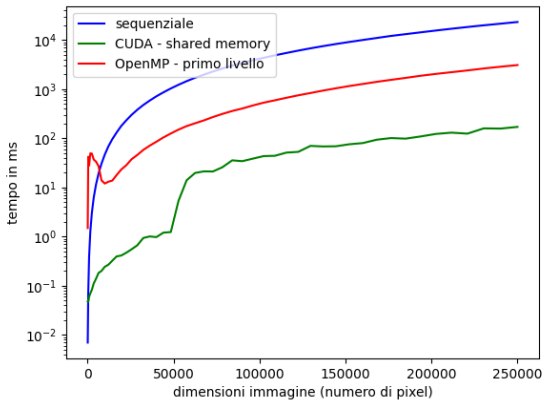

Figura 11: Tempo impiegato dall'esecuzione sequenziale e dalle due esecuzioni parallele al variare della grandezza dell'immagine.

Attraverso questo esperimento si può dunque notare come la versione parallelizzata attraverso CUDA sia molto più veloce non solo della versione sequenziale ma anche della versione parallelizzata attraverso OpenMP. Questo prova come la parallelizzazione compiuta attraverso la GPU sia superiore di quella compiuta attraverso la CPU.
Inoltre la versione OpenMP scelta (con cioè 100 thread) è più lenta dell'algoritmo sequenziale per dimensioni dell'immagine molto piccole: ciò è ovviamente dato dal fatto che il tempo di gestione di ben 100 thread non viene bilanciato dall'esecuzione velocizzata su così pochi pixel.