ALMA MATER STUDIORUM – UNIVERSITA' DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

PARALLELIZZAZIONE DI ALGORITMI DI ELABORAZIONE DI IMMAGINI

Elaborato in
High performance computing

Relatore
Moreno Marzolla

Presentata da
Ambra Albertini

Anno Accademico 2018/2019

Indice

1. Sommario

In questa tesi verificheremo l'efficacia della programmazione parallela utilizzata per implementare algoritmi il cui dominio può raggiungere grandi dimensioni, velocizzando i tempi di esecuzione e migliorando l'utilizzo dell'hardware a disposizione. Per eseguire questa verifica sono stati scelti alcuni algoritmi di creazione e modifica di immagini raster, poiché spesso algoritmi di questo tipo potrebbero contenere calcoli onerosi. Il primo scelto è l'algoritmo Diamond-Square, con cui è possibile creare proceduralmente immagini calcolando ogni pixel tramite due diversi passaggi che prenderanno in considerazione elementi dell'immagine già inizializzati per calcolare il valore di uno specifico pixel; verranno successivamente realizzate le operazioni di morfologia matematica (apertura e chiusura) su immagini raster a colori.

Per ognuno degli algoritmi viene proposta una versione seriale di riferimento e vengono descritte due versioni parallele usando due metodi di parallelizzazione diversi: una con OpenMP, sfruttando quindi la parallelizzazione offerta dal processore, e una con CUDA, utilizzando dunque la potenza di calcolo dei processori SIMD di una GPU (Graphics Processing Unit). Valuteremo le prestazioni delle implementazioni parallele e confronteremo quale delle tre sia la più conveniente a seconda dell'algoritmo in esame e in base al dominio di calcolo scelto, poiché sicuramente, modificando la grandezza del dominio, un metodo di implementazione potrebbe risultare più efficiente di un altro. Per poter effettuare queste osservazioni verrà creato un software utilizzando il motore grafico Unreal Engine con cui verrà creata un'interfaccia grafica per la visualizzazione delle immagini che si otterranno come risultato dell'esecuzione degli algoritmi presi in esame. Il linguaggio di programmazione che si userà sarà C++, con cui verranno create classi diverse per ogni versione dell'algoritmo; le immagini prodotte saranno inoltre esportabili dal programma in formato PNG per poter essere analizzate con più chiarezza in caso di necessità.

Oltre all'immagine prodotta, sull'interfaccia grafica verrà riportato il tempo di esecuzione dell'algoritmo per poterlo poi utilizzare per

calcolare l'efficienza delle varie versioni implementate. Questi ultimi calcoli mostreranno il miglioramento in prestazioni che si può ottenere parallelizzando l'algoritmo, ma verranno studiati anche diversi casi che potrebbero rendere una versione più vantaggiosa dell'altra. Per CUDA, ad esempio, verificheremo la differenza di prestazioni che si otterrà modificando il dominio dell'immagine; per OpenMP, invece, ci sarà la possibilità di modificare anche il numero di *threads* utilizzati per poter trovare sperimentalmente la quantità di *threads* da lanciare che porta ad ottenere una maggiore efficienza.

2. Introduzione

2.1. Programmazione parallela: perché e come utilizzarla

L'esecuzione di algoritmi computazionalmente onerosi richiede hardware sempre più potente, in grado di ridurre al minimo i tempi di esecuzione: per rispondere a questa esigenza ormai sono sempre più diffusi processori con un gran numero di core in grado di svolgere più calcoli contemporaneamente a seconda del tipo di architettura parallela adottata. I diversi tipi di architetture sono stati classificati nel 1966 nella tassonomia di Michael J. Flynn [1], in cui la suddivisione delle diverse architetture è stata eseguita tenendo conto dei flussi di istruzioni e dei dati che sono in grado di gestire contemporaneamente.

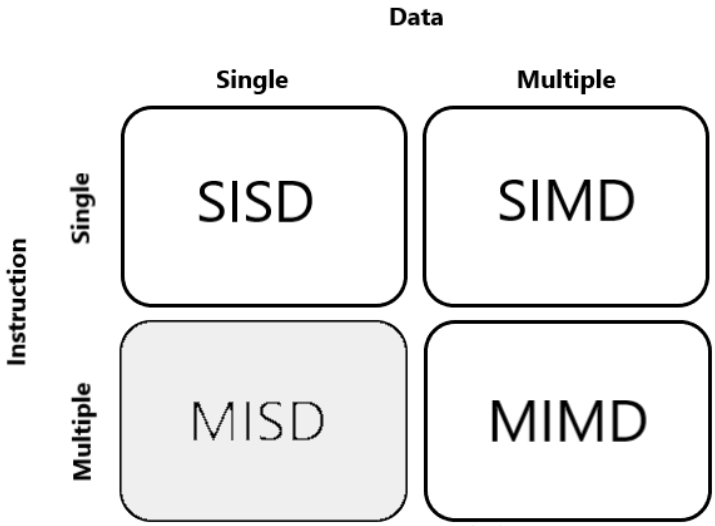

Figura 1 - Tassonomia di Flynn

Una rappresentazione visuale della tassonomia di Flynn è riportata nella Figura 1; in essa è possibile notare che le diverse classificazioni sono quattro: SISD (single instruction single data), SIMD (single instruction multiple data), MISD (multiple instruction single data) e MIMD (multiple instruction multiple data). A differenza delle altre, SISD è una famiglia di architetture di tipo seriale in cui può essere eseguito un solo flusso di istruzioni alla volta che operano su un unico flusso di dati. L'architettura SISD più conosciuta ed utilizzata è quella descritta nel 1945 da John von Neumann in *First draft of a report on the EDVAC* [2] e la sua caratteristica principale è quella di avere un'unica memoria in grado di contenere sia dati che istruzioni. Oltre a ciò presenta un processore (CPU: Central Processing Unit) composto da un'unità di calcolo (ALU: Arithmetic/Logic Unit) e un'unità di controllo, mentre un sottosistema di input/output comunica con le periferiche esterne.

In un'architettura SIMD lo stesso programma viene eseguito su più flussi di dati contemporaneamente; la maggior parte dei processori moderni mette a disposizione istruzioni SIMD con cui è possibile aumentare le prestazioni delle varie applicazioni vettorizzando i dati in input, ovvero eseguendo un'operazione su un vettore di più dati, anziché uno solo alla volta. Per la creazione di questi vettori ci si può affidare all'auto-vettorizzazione offerta dal compilatore, che però potrebbe generare codice non del tutto ottimizzato; alcuni compilatori, inoltre, supportano tipi di dato vettore definiti raggruppando dati dello stesso tipo, su cui è poi possibile eseguire operazioni aritmetiche di base. Questo tipo di vettorizzazione, tuttavia, benché indipendente dall'hardware sottostante, è strettamente legata al compilatore che definisce i tipi di dato vettore, mentre i *SIMD intrinsics* di Intel, sono funzioni C che dipendono dal processore sottostante, ma non dal compilatore utilizzato. Tali funzioni, infatti, permettono di accedere direttamente alle istruzioni SIMD messe a disposizione dal processore, quindi dipendono fortemente dal tipo di CPU utilizzata e da quale livello di istruzioni SIMD essa supporta [3]. L'architettura SIMD viene adottata anche dalle moderne GPU (Graphic Processing Unit): esse creano *threads* virtuali in base al numero di operazioni da eseguire, poi li dividono in blocchi contenenti tanti *threads* quante sono le unità di calcolo replicate all'interno del processore della GPU e mandano in esecuzione un blocco alla volta. Le architetture MIMD invece,

mandano in esecuzione più flussi di istruzioni, ognuno con il proprio flusso di dati indipendente; le architetture di questo tipo, inoltre, possono essere suddivise in due ulteriori sottocategorie: architetture a memoria condivisa e quelle a memoria distribuita. Nelle MIMD a memoria condivisa tutte le unità di esecuzione possono accedere alla stessa unità di memoria, rischiando quindi problemi di *race conditions*, poiché processori diversi potrebbero modificare lo stesso indirizzo in memoria. Nelle architetture a memoria distribuita, invece, ogni processore ha una memoria privata a cui solo lui può accedere ed è collegato agli altri per poter comunicare: questo fa sì che si evitino i problemi di concorrenza, tuttavia è necessario valutare bene le fasi di comunicazione perché lo scambio di messaggi potrebbe richiedere tempi elevati. A differenza delle altre architetture, infine, MISD è inutilizzata, poiché non ci sono vantaggi nel far eseguire più flussi di istruzioni sullo stesso flusso di dati.

Per quanto l'architettura di von Neumann descritta in precedenza sia l'architettura usata come standard per i calcolatori di tipo SISD, non è ancora stata scelta un'architettura parallela tipica: ogni produttore quindi utilizzerà un'architettura propria, in cui solamente i programmi scritti per essa potranno essere eseguiti, mentre quelli di altre architetture risulteranno incompatibili. CUDA, ad esempio, è una piattaforma di computazione parallela creata da NVIDIA, quindi è possibile programmare in CUDA solamente su GPU di questo produttore, poiché solo in esse è supportata. Esistono tuttavia standard di programmazione parallela che possono essere utilizzati anche su hardware diversi: OpenCL, per esempio, è un framework che offre un'astrazione hardware di basso livello per poter essere utilizzato su CPU, GPU e altri tipi di processori tramite la programmazione di *host* e *devices* utilizzando ANSI C e C++ [4]. Un ulteriore standard de facto utilizzabile su architetture diverse è OpenMP, che può essere usato in linea di principio su qualsiasi architettura MIMD a memoria condivisa, poiché si basa sul supporto dato dai compilatori C; non dipendendo quindi da una architettura specifica, programmi che fanno uso di OpenMP sono portabili ed eseguibili indipendentemente dall'hardware del computer.

2.1.1. Programmazione CUDA

Su un computer munito di scheda grafica NVIDIA è possibile creare programmi in grado di sfruttare la parallelizzazione offerta da CUDA. Per poter essere eseguito il codice dovrà essere compilato da NVCC, un compilatore proprietario di NVIDIA in grado di separare le parti di codice che dovranno essere mandate in esecuzione sulla GPU da quelle che invece saranno eseguite dalla CPU [5]. Per poter utilizzare CUDA, infatti, il codice sorgente deve essere formato da due parti: una di competenza della CPU (*host*) e l'altra della scheda grafica (*device*). La parte dell'*host* si fa carico di eseguire operazioni di input/output e trasferisce nella memoria globale della GPU i dati che dovranno essere utilizzati da quest'ultima per l'esecuzione. L'*host* può solamente trasferire dati verso il *device*, ma non è in grado di utilizzare i dati presenti sulla GPU senza prima averli trasferiti nella memoria della CPU. La parte che dovrà essere eseguita sul *device* è formata invece da funzioni particolari chiamate *kernel* che si distinguono dalle altre funzioni del codice sorgente tramite la parola chiave `__global__` inserita prima della definizione della funzione. Un *kernel* verrà invocato dal codice *host* che specificherà su quali dati dovrà lavorare passandogli i puntatori alla memoria globale della GPU come parametri e su quanti *threads* dovrà essere messo in esecuzione. Una chiamata ad un *kernel*, infatti, sarà formata nel modo seguente (Frammento 1):

```
nome_kernel<<<numero_blocchi, numero_threads>>>(…)
```

Frammento 1 - esempio di invocazione di un kernel CUDA

in cui `numero_blocchi` indica il numero di blocchi di *threads* da mandare in esecuzione, mentre `numero_threads` è il numero di *threads* per blocco. Durante la scelta del numero di *threads* e di blocchi da lanciare bisogna porre attenzione ai limiti della scheda grafica che si sta usando poiché un blocco può contenere solo un numero limitato di *threads* che in alcune GPU potrebbe essere anche solo 512. I *threads* che fanno parte di uno stesso blocco hanno la possibilità di comunicare tra loro e di utilizzare una memoria condivisa di dimensioni ridotte ma più veloce negli accessi rispetto alla memoria globale. Non è detto che i *threads* di un blocco vengano eseguiti contemporaneamente: esiste una ulteriore suddivisione all'interno del blocco chiamata *warp* formata dal massimo numero di *threads* che possono essere schedulati

contemporaneamente, in genere 32. Utilizzando *threads* dello stesso blocco ma di *warps* diversi, essi devono essere sincronizzati tra loro per evitare che uno acceda ad un indirizzo in memoria condivisa mentre un altro lo sta modificando, altrimenti si rischierebbero *race conditions* che comprometterebbero il risultato dell'esecuzione; tale sincronizzazione può essere fatta inserendo `__syncthreads()` per far sì che i *threads* restino bloccati fino al raggiungimento del punto di sincronizzazione da parte di tutti gli altri [6].

2.1.2. Programmazione OpenMP

OpenMP è un modello per la programmazione parallela su architetture a memoria condivisa che sfrutta i diversi processori di una CPU suddividendo tra di essi il carico di lavoro. OpenMP non è in grado di parallelizzare autonomamente il codice, ma offre la possibilità di creare regioni parallele all'interno del programma grazie a una serie di direttive specifiche con cui segnalare al compilatore che quella porzione di codice deve essere eseguita in maniera parallela. Durante la compilazione del codice è necessario abilitare il supporto a OpenMP aggiungendo alla linea di comando per la compilazione un'opzione particolare che dipende dal compilatore in uso: se, ad esempio, stiamo utilizzando GCC è necessario inserire l'opzione `-fopenmp`, mentre per Microsoft Visual C++ l'opzione è `/openmp`. Se il supporto non viene attivato oppure non è utilizzabile con un determinato compilatore, le direttive `#pragma` vengono ignorate e il codice viene eseguito sequenzialmente. La direttiva principale è `#pragma omp parallel`, con cui viene segnalato l'inizio della regione parallela. All'interno di essa si avrà modo di utilizzare il parallelismo dei dati oppure, dalla versione OpenMP 3.0, anche il parallelismo dei *tasks*. Nel primo tipo la parallelizzazione viene effettuata suddividendo i dati tra i vari processi, che eseguiranno quindi le stesse operazioni su dati diversi; la seconda tipologia, invece, viene eseguita suddividendo i compiti da svolgere, facendo lavorare tutti i processi sugli stessi dati ma non sulle stesse operazioni. Per poter utilizzare il parallelismo dei *tasks* all'interno della regione parallela devono essere inserite tante direttive `#pragma omp task` quanti sono i compiti che vogliamo svolgere in parallelo: la porzione di codice successiva a ognuna di queste direttive verrà poi assegnata a un *thread* per l'esecuzione [7].

Essendo una forma di programmazione parallela per architetture a memoria condivisa bisogna porre molta attenzione agli eventuali problemi di sincronizzazione: per questo motivo esistono le direttive `#pragma omp atomic` e `#pragma omp critical` in grado di evitare accessi contemporanei alla stessa zona di memoria. Tali direttive si distinguono dal fatto che `#pragma omp atomic` è in grado di proteggere una singola operazione, mentre `#pragma omp critical` crea regioni critiche composte anche da più istruzioni. Un'ulteriore direttiva di grande importanza è `#pragma omp for`, richiamata dentro alla regione parallela oppure fusa con `#pragma omp parallel` per poter così chiamare una sola direttiva (`#pragma omp parallel for`). Essa, seguita da un ciclo for, è in grado di dividere autonomamente le iterazioni del ciclo tra i vari *threads*, tuttavia in questo caso potrebbero verificarsi problemi di dipendenza dei dati, poiché le operazioni svolte all'interno di un'iterazione potrebbero dipendere dall'esito di quella precedente, causando quindi errori nel caso il *thread* che deve eseguire una determinata iterazione venga eseguito prima di quello che ha in carico quella precedente. Prima di iniziare la parallelizzazione di un ciclo, quindi, è necessario verificare che non ci siano dipendenze tra le varie iterazioni e se possibile rimuoverle modificando il ciclo opportunamente; nel caso non sia possibile togliere tali dipendenze il ciclo non può essere parallelizzato in maniera sicura [7].

2.2. Grafica raster

Esistono due tipi di immagine digitale: le immagini vettoriali e quelle raster. Il primo tipo è formato da un insieme di primitive geometriche e può essere visualizzata a schermo solamente dopo essere stata convertita in immagine raster, poiché uno schermo è in grado di mostrare solamente i pixel. Un'immagine raster, infatti, è una matrice di pixel (picture element) ognuno dei quali rappresenta un dato analogico misurato da un apposito sensore e successivamente campionato e quantizzato per ottenere la sua traduzione digitale. Il campionamento divide lo spazio dell'immagine in quadratini e per ognuno di essi esegue una media del colore presente in quella zona; ottenuto il valore medio del colore esso viene quantizzato riducendo le infinite tonalità reali a un numero finito di possibili colori che andranno a costituire il pixel. A differenza delle immagini raster, quelle vettoriali

sono basate su calcoli matematici come operazioni algebriche e trasformazioni geometriche: questo fa sì che un'immagine vettoriale sia scalabile a piacimento poiché il numero di pixel per la rappresentazione viene definito dopo l'operazione di scala per visualizzare al meglio la funzione geometrica ottenuta. Un'immagine raster, invece, ogni volta che viene scalata perde risoluzione poiché si tenta di visualizzare dati campionati con una certa scala su una più grande [8].

Un'immagine raster può essere in *grayscale* (scala di grigio), in bianco e nero (immagini binarie) o a colori; queste ultime possono fare riferimento a diversi tipi di formato. Le immagini in scala di grigio sono formate da pixel la cui dimensione in bit può variare a seconda di quanti livelli di grigio si vogliono utilizzare per definire le varie tonalità. Nella maggior parte dei casi tali livelli vengono specificati utilizzando 8 bit per pixel, ovvero ogni pixel che compone l'immagine ha una dimensione di 8 bit, tuttavia si potrebbe decidere di utilizzare anche pixel di grandezza maggiore. Definire le varie tonalità con 8 bit fa sì che ogni pixel possa prendere 2^8 valori distinti, ognuno dei quali, in base al formato grafico scelto, corrisponderà a un livello di grigio; nel formato utilizzato in questa tesi 0 corrisponde al nero e 255 al bianco, mentre i valori intermedi sono le varie tonalità di grigio possibili. Per le immagini a colori, invece, la struttura di un pixel dipende dal formato scelto: esse, infatti, possono essere in formato RGB oppure HS*. Per le immagini in formato RGB i vari colori sono dati dalla combinazione dei valori contenuti in canali distinti, solitamente uno per il rosso, uno per il verde e uno per il blu. Tali canali possono essere visti come immagini in scala di grigio, quindi la loro grandezza in bit dipende da quante tonalità distinte devono essere definite: nei formati grafici più utilizzati ogni canale è composto da 8 bit per ogni pixel, tuttavia è possibile usare anche dimensioni maggiori per avere a disposizione più tonalità. Dato che i canali colore citati sono tre, ogni pixel sarà quindi formato da 24 bit, ma per alcune immagini possono essere anche 32 poiché oltre ai canali rosso, verde e blu ne è presente uno per l'eventuale trasparenza: nel formato grafico utilizzato in questa tesi il valore che renderà trasparente il pixel è 0, mentre tutti gli altri valori lo faranno risultare sempre più nitido fino alla totale assenza di trasparenza a 255. In presenza di trasparenza, quindi, verrà aggiunto all'immagine RGB un canale A (*Alpha*) e si parlerà così di formato RGBA. Il formato HS*,

invece, è in realtà un insieme di più modelli, di cui il più usato è HSL (o HSB), che codificano un colore in base alle caratteristiche percepite dall'essere umano: tinta (*Hue*), saturazione (*Saturation*) e luminosità (*Lightness* o *Brightness*) [8].

Dato che un'immagine raster può essere considerata una matrice di pixel, alcune operazioni di modifica possono essere anche molto onerose, poiché è necessario scorrere tutti i valori della matrice che potrebbero essere anche molto numerosi: per questo motivo l'implementazione parallela di algoritmi di modifica può migliorare anche di molto le prestazioni e l'efficienza dell'esecuzione. In questa tesi verranno usate immagini a colori in formato RGBA per poter così salvare il risultato in un file PNG.

2.3. Unreal Engine: studio e preparazione dell'ambiente di sviluppo

Unreal Engine 4 è un motore grafico di Epic Games su cui è possibile programmare in C++ oppure tramite Blueprint, un linguaggio di scripting visuale orientato alla programmazione ad oggetti [9]. Esso è stato scelto come ambiente di sviluppo per poter sfruttare le sue capacità grafiche per la creazione di interfacce grafiche e la possibilità di scrivere codice in C++, con cui è possibile poi programmare utilizzando la libreria *omp.h* per OpenMP. Come già detto, tuttavia, non basta solo includere la libreria per poter utilizzare questo metodo di parallelizzazione, ma deve essere abilitato con un comando durante la compilazione. Purtroppo, essendo un programma Unreal Engine, il Makefile con cui vengono compilati i progetti fa parte del motore grafico stesso, quindi per poterlo modificare è necessario ricompilare tutto Unreal Engine. Se, tuttavia, quest'ultimo è stato scaricato da Epic Games Launcher [1], benché sia possibile modificare il codice sorgente, non può poi essere ricompilato.

Per poter modificare e utilizzare il codice di Unreal Engine è possibile scaricare il codice sorgente dal repository GitHub di Epic Games (https://github.com/EpicGames/UnrealEngine), in cui si può accedere solamente se si possiede un profilo Unreal Engine e uno di GitHub connessi tra loro. All'interno del repository si possono trovare diverse

[1] Epic Games Launcher è il programma da cui è possibile scaricare le diverse versioni di Unreal Engine.

versioni del motore grafico: al momento è stata raggiunta la versione 4.24.2, ma per questa tesi è stata utilizzata la versione 4.22. Una volta scaricato il codice della versione desiderata è possibile creare la soluzione di Visual Studio tramite il file eseguibile GenerateProjectFiles.bat [10]. Per poter utilizzare OpenMP è necessario modificare il file VCToolChain.cs aggiungendo al metodo `AppendCLArguments_Global` la seguente istruzione riportata nel Frammento 2:

```
Arguments.Add("/openmp");
```

Frammento 2 - linea di codice con cui inserire parametri alla riga di comando per la compilazione di un progetto Unreal Engine

All'interno della parentesi è stata inserita la stringa corrispondente al parametro da passare alla riga di comando durante la compilazione del programma OpenMP utilizzando il compilatore Microsoft Visual C++ di Visual Studio; poiché però Unreal Engine è multipiattaforma, all'interno del metodo `AppendCLArguments_Global` sono presenti delle istruzioni condizionali *if* in cui viene controllato il tipo di compilatore in utilizzo: per ognuno di essi è quindi possibile aggiungere l'istruzione sopra riportata modificando opportunamente la stringa con il comando giusto per il compilatore in uso. Una volta aggiunta per tutti i compilatori che si vogliono utilizzare è possibile compilare il motore grafico ed eseguirlo per iniziare a creare progetti con esso [11]. Una volta eseguiti questi passaggi e creato il progetto Unreal Engine, per poter utilizzare OpenMP basta semplicemente inserire `#include <omp.h>` tra le inclusioni della classe C++ che conterrà i metodi con cui verranno implementati gli algoritmi.

Per poter utilizzare CUDA, invece, non serve modificare nuovamente il motore grafico perché in realtà è necessario utilizzare come compilatore NVCC, non supportato da Unreal Engine: è stato scelto quindi di creare una libreria statica C++ a parte, compilata con il compilatore corretto e richiamata all'interno del progetto Unreal Engine nelle classi che devono usare CUDA. Per poter utilizzare librerie esterne bisogna modificare il file NomeProgetto.Build.cs inserendo nel costruttore della classe:

```
PublicAdditionalLibraries.Add(
    “percorso_libreria/libreria.lib”);
PublicIncludePaths.Add(“percorso_headers”);
```

Frammento 3 - linee di codice per l'inclusione di librerie esterne in un progetto Unreal Engine

per ogni libreria che si vuole utilizzare [12]. Per l’uso di CUDA, oltre alla libreria da noi creata con i *kernel* per l’esecuzione degli algoritmi deve essere aggiunta anche la libreria statica `cudart_static.lib` poiché se non fosse inclusa non sarebbe possibile utilizzare le funzioni messe a disposizione da CUDA.

2.4. Preparazione dell’interfaccia grafica del programma

Nelle seguenti figure viene mostrata l’interfaccia grafica realizzata per eseguire i due diversi algoritmi: essa è formata da due *widgets* implementate usando il linguaggio di scripting Blueprint. Per programmare una *blueprint widget* sono disponibili due visuali diverse: una chiamata *Designer*, in cui viene progettata la parte visuale dell’interfaccia, e una *Graph*, dentro alla quale viene implementata la logica tramite grafi e funzioni Blueprint [13].

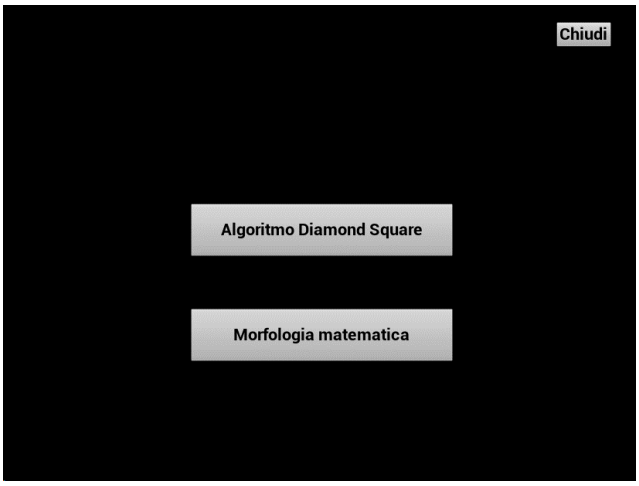

Figura 2 - Widget iniziale del programma

La prima interfaccia, mostrata in Figura 2, è stata chiamata `MainWidget` e contiene semplicemente tre pulsanti: il primo, in alto a destra, è utilizzato per chiudere il programma tramite la funzione Blueprint `QuitGame` [14], mentre gli altri due richiamano una funzione creata internamente alla *widget* chiamata `CreateTextureEditor` (Figura 3).

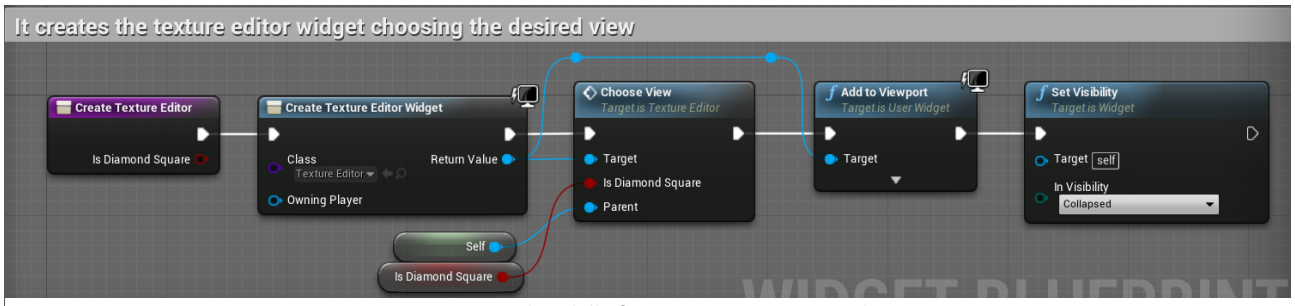

Figura 3 - Codice della funzione CreateTextureEditor

In questo frammento di codice viene utilizzata la funzione Blueprint `CreateWidget` per creare un'istanza della seconda *widget* del programma, chiamata `TextureEditor` (Figura 4), poi viene richiamato per quest'ultima l'evento `ChooseView`, con cui viene specificato se l'istanza appena creata deve eseguire l'algoritmo Diamond-Square oppure operazioni di morfologia matematica. Il parametro `IsDiamondSquare`, infatti, è un booleano che viene utilizzato in `ChooseView` per rendere visibili solamente gli elementi di `TextureEditor` utili per l'esecuzione dell'algoritmo scelto. A seguito della chiamata di tale evento l'istanza della nuova *widget* viene mostrata a schermo tramite `AddToViewport` [15], mentre la `MainWidget` viene resa invisibile tramite `SetVisibility` [16]. La presenza di `ChooseView` è necessaria perché la *widget* `TextureEditor` non è specifica per un solo algoritmo, ma contiene i campi utili per l'esecuzione di entrambi: nella colonna a sinistra, ad esempio, è possibile osservare la presenza del testo "Scegliere X" seguito da una *textbox*. Essa dovrà essere riempita con il valore che si vuole dare all'esponente dell'equazione $2^x + 1$ che indica il lato della texture creata con l'algoritmo Diamond-Square; il testo "Dimensione" sottostante conterrà invece il risultato di tale equazione. I campi appena descritti non servono per le operazioni di morfologia matematica, mentre i pulsanti "Carica immagine", "Apertura" e "Chiusura" sono utilizzati solo per l'esecuzione di esse: il primo dà la possibilità di scegliere un'immagine da filesystem, poiché le operazioni di morfologia matematica vengono eseguite partendo da un'immagine preesistente; gli altri due pulsanti, invece, permettono di scegliere se eseguire un'apertura o una chiusura sull'immagine di input. Tutti i campi sottostanti sono comuni a entrambe le esecuzioni perché le *combobox* servono per specificare quale versione utilizzare, i pulsanti

per far partire l'esecuzione e per salvare il risultato in PNG, mentre il campo "Tempo" conterrà il tempo impiegato per eseguire l'algoritmo scelto; la *textbox* "Thread", infine, servirà per specificare il numero di *threads* con cui eseguire la versione parallela con OpenMP.

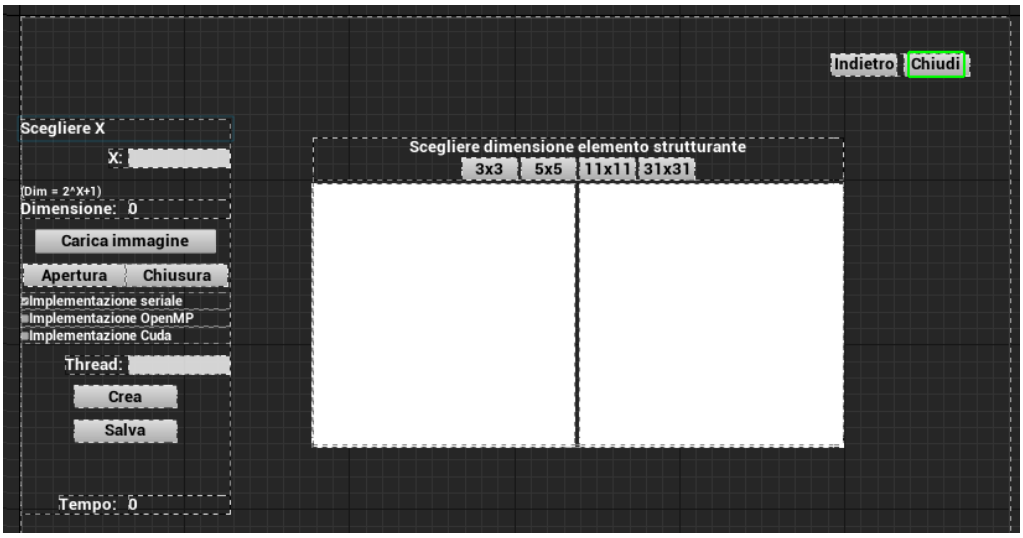

Figura 4 - Visuale di progettazione dell'interfaccia TextureEditor

Al centro sono presenti due texture che nella progettazione risultano bianche perché verranno inizializzate a tempo di esecuzione: quella più a sinistra conterrà l'immagine di input scelta al click del pulsante "Carica immagine" (solo per operazioni di morfologia matematica), mentre quella più a destra sarà la texture di output caricata al click del pulsante "Crea" (per entrambi gli algoritmi); sopra di esse, inoltre, per eseguire l'apertura o la chiusura, è stata inserita una sezione con diversi pulsanti che indicano la dimensione dell'elemento strutturante da utilizzare. Al click di uno di questi pulsanti viene comunicato alla classe che esegue le operazioni di morfologia quale elemento strutturante usare tra quelli messi a disposizione, perciò essi non sono necessari all'esecuzione dell'algortimo Diamond-Square. In alto a destra, infine, sono presenti un pulsante "Indietro" per tornare alla `MainWidget`, e un pulsante "Chiudi" per chiudere il programma, equivalente al pulsante già presente nell'altra *widget* citata. Il pulsante "Indietro" renderà nuovamente visibile l'istanza di `MainWidget` creata all'avvio del programma e rimuoverà quella di `TextureEditor` dalla scena tramite la funzione Blueprint `RemoveFromParent` [17]: in questo modo,

mentre ad ogni esecuzione del programma è presente una sola istanza di `MainWidget`, poiché viene nascosta e resa visibile a piacimento, ogni volta che si decide di non visualizzare più `TextureEditor`, tale istanza viene definitivamente rimossa e una nuova verrà ricreata al click di uno dei due pulsanti dell'interfaccia principale. Il pulsante "Crea" viene attivato solamente una volta scelto X, nel caso si voglia implementare una versione seriale o con CUDA, mentre con OpenMP è necessario anche specificare il numero di *threads* con cui eseguire tale versione parallela. Per le operazioni di morfologia matematica, tuttavia, le *combobox* e la *textbox* del numero di *threads* sono disattivate finché non viene caricata l'immagine di input, scelto quale tipo di operazione svolgere e lo strutturante da utilizzare, poiché senza queste informazioni non è possibile eseguire le operazioni richieste.

All'interno di `TextureEditor` vengono invocate funzioni scritte in C++ nelle classi `TextureCreator` e `TextureUtilities`; queste classi derivano dalla classe `BlueprintFunctionLibrary` [18], che può essere usata per contenere funzioni statiche non appartenenti a una specifica classe Blueprint, quindi utilizzabili in qualsiasi contesto all'interno del programma. Le funzioni di `TextureCreator` e `TextureUtilities`, quindi, oltre che ad essere richiamate dentro `TextureEditor`, possono essere utilizzate anche in altre classi C++ o Blueprint, in altre widgets oppure anche in una mappa, all'interno del suo livello Blueprint: esso è la parte della mappa in cui è possibile scatenare eventi tramite grafi Blueprint [19]. Nel progetto in esame, ad esempio, il livello Blueprint della mappa è utilizzato per creare l'istanza di `MainWidget` allo scatenarsi dell'evento `BeginPlay` all'avvio del programma. Per far sì che classi C++ siano utilizzabili anche via Blueprint è necessario che siano etichettate con la macro UFUNCTION, con cui è possibile dare alla funzione informazioni aggiuntive tramite diversi possibili specificatori [20].

`TextureCreator`, ad esempio, contiene le seguenti funzioni pubbliche:

```
UFUNCTION(BlueprintCallable,
    Category = "DiamondSquare")
    static UTexture2D* CreateProceduralTexture(
          ImplementationType implementationType,
          int size, int threadNumber,
          float &executionTime);

UFUNCTION(BlueprintCallable,
    Category = "MathematicalMorphology")
    static UTexture2D* LoadImage();

UFUNCTION(BlueprintCallable,
    Category = "MathematicalMorphology")
    static UTexture2D* ExecuteMMOperation(
          ImplementationType implementationType,
          int threadNumber, float &executionTime,
          bool isOpening, int structElemSize);
```

Frammento 4 - classi publiche definite in TextureCreator

Come si può notare dal Frammento 4, le tre funzioni sono utilizzabili via Blueprint grazie allo specificatore *BlueprintCallable* all'interno della macro UFUNCTION, mentre *Category* indica in quale categoria è possibile visualizzare la funzione all'interno del menù di azioni disponibili all'interno di un grafo Blueprint. La prima funzione riportata servirà per creare un'immagine tramite l'algoritmo Diamond-Square, mentre l'ultima eseguirà operazioni di morfologia matematica sull'immagine caricata da `LoadImage()`. In `CreateProceduralTexture(…)` e `ExecuteMMOperation(…)`, inoltre, è presente un parametro di tipo `ImplementationType`: tale tipo di dato è una enumerazione con cui viene specificata la versione dell'algoritmo scelto che si vuole eseguire, ed è definita in `TextureCreator` nel modo seguente:

```
UENUM(BlueprintType)
enum ImplementationType
{
       IT_Serial UMETA(DisplayName = "Serial"),
       IT_OpenMP UMETA(DisplayName = "OpenMP"),
       IT_Cuda UMETA(DisplayName = "Cuda"),
};
```

Frammento 5 - tipo di dato enumerazione definito in TextureEditor per i vari tipi di implementazione possibili

Anche per le enumerazioni, se è necessario che siano utilizzabili via Blueprint, esiste una macro UENUM che come UFUNCTION accetta specificatori; in questo caso viene utilizzato *BlueprintType* per far sì che la enumerazione venga vista come un tipo di dato Blueprint, mentre ogni valore presente all'interno di essa presenta la macro UMETA con lo specificatore *DisplayName*. Quest'ultimo viene utilizzato per definire per quel determinato valore il nome che verrà visualizzato nel codice Blueprint. In `TextureUtilities`, invece, l'unica funzione visibile in Blueprint è la seguente:

```
UFUNCTION(BlueprintCallable,
    Category = "TextureUtilities")
    static void SaveToPNG(AlgorithmType algorithm);
```

Frammento 6 - classe utilizzabile via Blueprint definita in TextureUtilities

Tale funzione salva l'immagine ottenuta come risultato in formato PNG all'interno della cartella *OutputImages* presente nella cartella del programma; il parametro `AlgorithmType` è una enumerazione con cui è possibile specificare da quale algoritmo è stata creata l'immagine che si vuole salvare.

```
UENUM(BlueprintType)
enum AlgorithmType
{
    AT_DiamondSquare
          UMETA(DisplayName="DiamondSquare"),
    AT_Opening UMETA(DisplayName="Opening"),
    AT_Closing UMETA(DisplayName="Closing")
};
```

Frammento 7 – Tipo di dato enumerazione definito in TextureUtilities per specificare da quale algoritmo è stata ottenuta l'immagine

Anche questa enumerazione è resa visibile nel codice Blueprint per poterla utilizzare nella chiamata a `SaveToPNG(…)` eseguita al click del pulsante "Salva" in `TextureEditor`.

Le due funzioni di `TextureCreator` riportate nel Frammento 4, `CreateProceduralTexture(…)` e `ExecuteMMOperation(…)`, utilizzano al loro interno oggetti di classi C++ create appositamente per l'esecuzione degli algoritmi richiesti; a differenza delle due classi descritte precedentemente, queste nuove classi non necessitano di derivare da una specifica classe definita in Unreal Engine, poiché i

metodi al loro interno non devono essere visibili in Blueprint. Esse sono, quindi, semplici classi C++ per cui è possibile creare istanze all'interno di altre classi. Avendo tuttavia la possibilità di eseguire lo stesso algoritmo utilizzando diverse versioni di implementazione, invece di creare una sola classe per algoritmo da implementare, è stato scelto di crearne una per ogni versione, derivabili da una classe base virtuale contenente le caratteristiche comuni. In questo modo, all'interno delle funzioni di `TextureCreator` verrà definito un oggetto della classe base, inizializzato poi come oggetto di una delle classi derivate a seconda del valore del parametro di tipo `ImplementationType`. Nel Frammento 8 seguente viene portato come esempio un frammento di codice estratto dalla funzione `CreateProceduralTexture(…)`.

```
…
switch (implementationType)
{
    case ImplementationType::IT_Serial:
          implementation =
                new SerialDiamondSquare(matrixSize);
          break;
    case ImplementationType::IT_OpenMP:
          implementation =
                new OpenMPDiamondSquare(matrixSize,
                threadNumber);
          break;
    case ImplementationType::IT_Cuda:
          implementation =
                new CudaDiamondSquare(matrixSize);
          break;
          default:
                break;
}
start = clock();
matrix = implementation->ExecuteDiamondSquare();
end = clock();
…
```

Frammento 8 - frammento di codice estratto di CreateProceduralTexture(...)

In esso viene riportato il costrutto condizionale `switch` con cui, in base al valore dato al parametro `implementationType`, viene inizializzata

la variabile `implementation`, precedentemente definita come oggetto della classe base. Per ogni tipo di implementazione tale oggetto sarà istanza di una classe diversa scelta tra `SerialDiamondSquare`, `OpenMPDiamondSquare` o `CudaDiamondSquare`, mentre l'esecuzione vera e propria dell'algoritmo verrà eseguita all'esterno dello switch. Una volta inizializzata la variabile implementation, infatti, la chiamata al metodo `ExecuteDiamondSquare()` viene eseguita tramite polimorfismo indipendentemente dalla classe derivata usata, poiché tale metodo come metodo virtuale della classe base. Questo fa sì che quando un'istanza di tale classe lo richiama, verrà eseguita l'implementazione offerta dalla classe derivata effettivamente utilizzata. Dal Frammento 8 è possibile notare anche la presenza delle due variabili `start` ed `end`, precedentemente definite all'interno della funzione. Esse vengono inizializzate tramite l'invocazione di `clock()` prima e dopo l'esecuzione dell'algoritmo perché in questo modo è possibile calcolare i tempi di esecuzione di esso, ma si può osservare che né l'inizializzazione dell'oggetto `implementation`, né la creazione della texture finale per la visualizzazione rientrano nel calcolo di questi tempi: i tempi di esecuzione riportati, quindi, faranno riferimento solamente all'esecuzione vera e propria dell'algoritmo che riporterà come risultato una matrice di byte, mentre la texture verrà creata a partire da questo risultato solamente in seguito. Tutte queste considerazioni possono essere fatte anche analizzando il codice della funzione `ExecuteMMOperation(…)`, poiché è stato impostato in maniera simile a quello di `CreateProceduralTexture(…)`, sebbene l'implementazione dell'algoritmo di morfologia matematica sia contenuta in altre classi, di cui si discuterà in seguito.

3. Algoritmo Diamond-Square

3.1. Descrizione dell'algoritmo

L'algoritmo Diamond-Square è un algoritmo di creazione di texture procedurali ideato da Fournier, Fussell e Carpenter nel 1982 [21], utile per la creazione di *heightmap* per la generazione di terreni in computer grafica. Una *heightmap* è un'immagine raster usata per dare profondità e irregolarità ai terreni e ai modelli 3D: in base al colore del pixel, infatti, il punto corrispondente viene rappresentato più vicino o più lontano dalla superficie. Essa può essere sia in scala di grigio, come quelle che verranno create in questa tesi, sia a colori a seconda della precisione che si vuole ottenere, poiché in base alla grandezza del terreno che si vuole creare potrebbero essere necessarie più sfumature di colore per ottenere un'irregolarità accettabile. Per *heightmap* in scala di grigio, infatti, le diverse differenze di altezza dal terreno variano da 0, dato dal colore nero e corrispondente al valore più vicino al suolo, a 255, che corrisponde al bianco. Se il terreno su cui si vuole applicare la *heightmap* è di grandi dimensioni, un intervallo di soli 256 valori in certi punti potrebbe produrre differenze di altezza molto evidenti che possono essere evitate aumentando l'intervallo di valori possibili, passando quindi a una *heightmap* a colori [22].

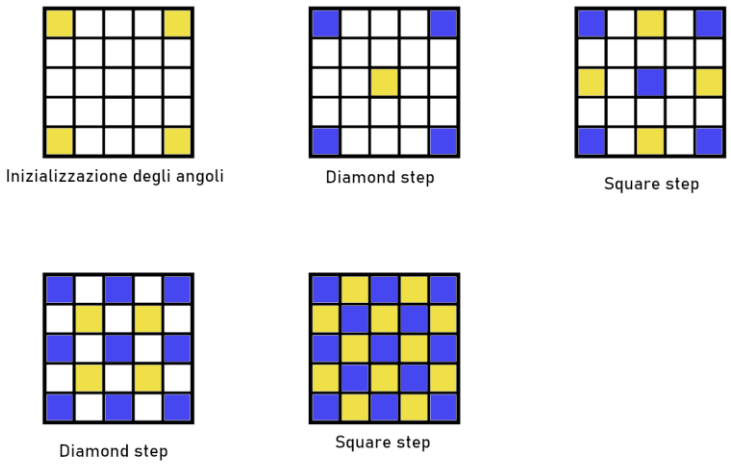

Figura 5 - Esecuzione dell'algoritmo Diamond-Square

Con l'algoritmo Diamond-Square è possibile creare immagini quadrate con lato $2^n + 1$ inizializzando i valori dei pixel tramite due passi, *diamond step* e *square step*, ripetuti ricorsivamente fino al completamento dell'immagine. Nella figura precedente (Figura 5) viene mostrato graficamente come vengono utilizzati i due passi per il completamento di una matrice $5x5$. Una volta inizializzati gli angoli, essi vengono usati come punti di partenza per il primo *diamond step*, con cui viene inizializzato il valore centrale della matrice. Una volta ottenuto anche questo valore viene eseguito uno *square step* che inizializza i valori centrali della prima e dell'ultima riga e della prima e ultima colonna. I passaggi vengono iterati finché tutta l'immagine non sarà completa.

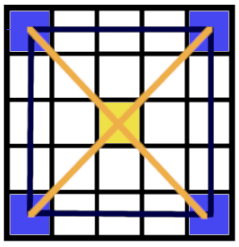

Figura 6 - Inizializzazione del valore centrale di un quadrato della matrice tramite Diamond Step

In generale, come mostrato in Figura 6, durante i *diamond steps* vengono inizializzate le celle centrali di ogni quadrato presente all'interno della matrice utilizzando la media dei valori degli angoli e un valore casuale il cui seme deve essere ridotto ad ogni iterazione. Negli *square steps*, invece, vengono considerati i valori agli angoli di ogni diamante della matrice per calcolare il valore al centro di tale diamante, come si può notare in Figura 7. Da queste ultime due immagini si può osservare che, mentre gli angoli di un quadrato cadono sempre all'interno del dominio della matrice, quelli di un diamante potrebbero anche essere esterni: durante lo *square step*, dunque, si rischia di accedere a indirizzi di memoria non facenti parte della matrice. Tale problema può essere evitato adottando un dominio ciclico, facendo quindi in modo che l'angolo all'esterno sia invece preso da una *ghost area* contenente i valori delle righe e delle colonne dall'altro lato della matrice. Una seconda soluzione adottabile consiste nel calcolare i valori

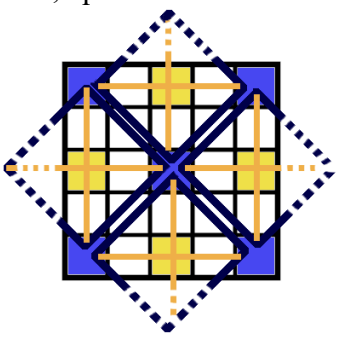

Figura 7 - Inizializzazione del valore centrale dei diamanti della matrice tramite Square Step

centrali dei diamanti con la media dei soli tre angoli esistenti, evitando in questo modo l'utilizzo di *ghost cells*. Purtroppo, considerando solamente tre angoli, i valori calcolati tramite *square step* potrebbero produrre righe verticali e orizzontali chiamate artefatti perché vengono calcolati utilizzando meno informazioni sul loro contorno di quelle necessarie per ottenere il valore corretto. Questo problema è stato discusso anche da Gavin S. P. Miller nel 1986 [23].

3.2. Sviluppo e implementazione

Per poter utilizzare una *ghost area* è necessario allocare in memoria, oltre allo spazio per la matrice, quello che dovrà contenere le righe e le colonne aggiuntive poste prima e dopo quelle della matrice; quest'ultima sarà così circondata da celle "fantasma" che non fanno realmente parte del dominio ma che conterranno una copia delle celle della matrice sul lato opposto per simulare la periodicità del dominio. In questo contesto, tuttavia, il numero di righe o colonne in più da allocare dipende dalla dimensione della matrice: la Figura 8, ad esempio, mostra la dimensione della *ghost area* necessaria per calcolare il valore della cella più a destra del primo *square step*. Per semplicità viene mostrata solamente la *ghost area* a destra della matrice, mentre all'interno delle celle si può notare la presenza di sigle: **I** indica i valori impostati inizialmente, **1d** sono invece le celle inizializzate durante il primo *diamond step*, mentre **1s** quelle ottenute con il primo *square step*. Nell'esempio viene mostrata una matrice $9x9$ e la *ghost area* di destra è formata da quattro colonne; per il lato sinistro sarà necessaria la stessa quantità di colonne, mentre quattro righe saranno usate sia per il lato superiore che inferiore. Grazie a queste considerazioni si può comprendere che per una matrice $2^n + 1$ il numero di righe e di colonne aggiuntive necessarie è 2^{n-1} su ogni lato, poiché esso rappresenta la distanza delle celle di uno dei bordi dal centro della matrice. Per i passi successivi, tuttavia, non serve un numero così elevato di *ghost cells* perché la distanza tra le celle da inizializzare e quelle utilizzate per calcolarne il valore viene dimezzato

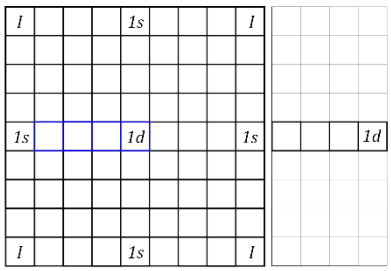

Figura 8 - Calcolo di un valore laterale tramite Square Step utilizzando Ghost Cells (prima iterazione dell'algoritmo)

ad ogni passaggio: per il secondo *square step*, quindi, per ogni lato basteranno 2^{n-2} righe o colonne, lasciando inutilizzate le restanti 2^{n-2}. In Figura 9, infatti, viene mostrato il secondo *square step* eseguito sulla matrice $9x9$ dell'esempio precedente; in essa sono state aggiunte le sigle **2d** per i valori calcolati durante il secondo *diamond step* e **2s** per quelli ottenuti dal secondo *square step*.

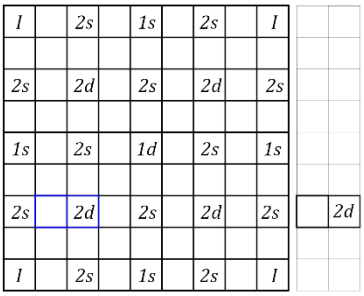

Figura 9 - Calcolo di un valore laterale tramite Square Step utilizzando Ghost Cells (seconda iterazione dell'algoritmo)

Avendo, quindi, necessità di un gran numero di righe e colonne aggiuntive, soprattutto per la prima iterazione, l'utilizzo di una *ghost area* potrebbe risultare poco conveniente perché per ogni matrice dovrebbe essere allocato uno spazio circa tre volte più grande della dimensione effettiva del dominio: data una matrice $(2^n + 1)x(2^n + 1)$, infatti la *ghost area* sarebbe una matrice $2^{n-1}x(2^n + 1)$ su ogni lato, quindi:

Equazione 1 - calcolo della dimensione della ghost area

$$ghost\ area = 4x2^{n-1}x(2^n + 1) = 2^n x(2^n + 1)x2$$
$$dominio = (2^n + 1)x(2^n + 1) = (2^n + 1)x2^n + (2^n + 1)$$
$$ghost\ area = 2\ x\ dominio - 2(2^n + 1)$$

ovvero circa due volte in più della grandezza del dominio, meno una riga e una colonna. Per le versioni implementate di questo algoritmo, quindi, è stato scelto di mantenere finito il dominio, eseguendo quindi le medie dei valori laterali utilizzando solamente i tre angoli esistenti.

Come accennato in precedenza in **2.4 Preparazione dell'interfaccia grafica del programma** (a pagina 13), per implementare le diverse versioni dell'algoritmo sono state create diverse classi, tra cui `DiamondSquareAlgorithm` è la base delle altre e la cui definizione viene riportata nel Frammento 9 seguente.

```
class HPCIMAGEPROCESSING_API DiamondSquareAlgorithm
{
    public:
          DiamondSquareAlgorithm(int size);
          virtual ~DiamondSquareAlgorithm();
          virtual uint8* ExecuteDiamondSquare() = 0;
    protected:
          virtual void DiamondSquare(int matrixSize,
                int maxValue) = 0;
          virtual void DiamondStep(int row,
                int column, int adding,
                int maxValue) = 0;
          virtual void SquareStep(int row,
                int column, int adding,
                int maxValue) = 0;

          uint8* image;
          int size;
};
```

Frammento 9 - Dichiarazione della classe DiamondSquareAlgorithm

Dalla dichiarazione appena riportata è possibile notare che `DiamondSquareAlgorithm` è una classe astratta: questa condizione è data dal fatto che i metodi della classe, oltre ad essere contrassegnati con la parola chiave `virtual`, sono anche inizializzati a 0, poiché la sola presenza di tale parola chiave non basta a definire una classe astratta. La parola `virtual`, infatti, viene solitamente usata nei metodi di classi base per far sì che possano essere ridefiniti tramite *override* nelle classi derivate da essa: se un metodo virtuale viene invocato tramite polimorfismo da un oggetto della classe base inizializzato come oggetto della classe derivata, sarà eseguita l'implementazione data dalla classe derivata, mentre se non fosse provvisto di parola chiave `virtual`, l'unica implementazione utilizzabile sarebbe quella della classe di definizione, ovvero quella della classe base. Qualsiasi classe può contenere metodi virtuali perché potrebbe essere utilizzata come base per altre classi, ma non per questo è considerata classe astratta. Quest'ultimo tipo di classe, invece, si ha quando, oltre a contenere metodi virtuali, almeno uno di questi è inizializzato a 0, ovvero non ha nessuna implementazione all'interno della classe base. Nelle classi derivate è necessario che tutti questi metodi virtuali, detti puri, siano

ridefiniti, altrimenti anch'esse sarebbero considerate astratte e non sarebbe possibile istanziare oggetti di quella determinata classe.

3.2.1. Versione seriale

La prima classe derivata da `DiamondSquareAlgorithm` è `SerialDiamondSquare`; essa contiene l'implementazione della versione seriale di riferimento dell'algoritmo Diamond-Square e la sua definizione è riportata nel Frammento 10.

```
class HPCIMAGEPROCESSING_API SerialDiamondSquare
    : public DiamondSquareAlgorithm
{
    public:
          SerialDiamondSquare(int size)
                : DiamondSquareAlgorithm(size) {}
          ~SerialDiamondSquare() {}
          uint8* ExecuteDiamondSquare();
    protected:
          void DiamondSquare(int matrixSize,
                int maxValue);
          void DiamondStep(int row, int column,
                int adding, int maxValue);
          void SquareStep(int row, int column,
                int adding, int maxValue);
};
```

Frammento 10 - definizione della classe SerialDiamondSquare

Da tale frammento si può notare che il costruttore della classe richiama quello della classe base ma non contiene alcuna istruzione poiché non ci sono variabili in più rispetto alla base che devono essere inizializzate. Avendo solamente richiamato il costruttore base, dunque, anche il distruttore sarà vuoto, perché grazie al polimorfismo andrà a richiamare quello della classe base che sarà sufficiente per la liberazione dell'eventuale memoria allocata per i campi della classe. Nel costruttore di `DiamondSquareAlgorithm`, infatti, dovrà essere allocata memoria per il puntatore a interi a 8 bit che conterrà i byte dell'immagine risultato (campo `image` nel Frammento 9), mentre nel distruttore tale memoria dovrà essere liberata.

All'interno di `ExecuteDiamondSquare()` verranno inizializzati i valori degli angoli dell'immagine con valori casuali presi da un

intervallo di 256 valori, da 0 a 255, ovvero le varie tonalità di colore che è possibile dare a un pixel di 8 bit. In seguito, verrà richiamato il metodo ricorsivo `DiamondSquare(int matrixSize, int maxValue)` in cui `matrixSize` inizialmente è 2^n, l'ultimo indice delle righe o delle colonne della matrice, mentre `maxValue` è il valore massimo che può avere il valore casuale, poiché come si è già detto ad ogni iterazione il seme deve diminuire. Nel seguente Frammento 11 viene riportata una porzione codice di questa funzione:

```
…
if (matrixSize > 1)
{
    //Diamond step
    for (i = 0; i < last; i += matrixSize)
    {
          for (j = 0; j < last;
                j += matrixSize)
          {
                this->DiamondStep(i, j, half,
                      maxValue);
          }
    }
    //Square step
    for (i = 0; i < this->size; i += half)
    {
          if (i%matrixSize == 0)
          {
                startIndex = half;
                endSquare = last;
          }
          else
          {
                startIndex = 0;
                endSquare = this->size;
          }
          for (j = startIndex; j < endSquare;
                j += matrixSize)
          {
                this->SquareStep(i, j, half,
                      maxValue);
          }
    }
    this->DiamondSquare(half, maxValue / 2);
```

```
}
…
```

Frammento 11 – frammento di codice del metodo DiamondSquare

Finché `matrixSize` è maggiore di 1 vengono eseguiti *diamond steps* e *square steps* iterati tante volte quanti sono i quadrati e i diamanti all'interno della matrice. Durante il *Diamond step* gli elementi della matrice vengono scorsi sia su righe che su colonne dall'indice 0 fino al penultimo, $2^n - 1$, scorrendone `matrixSize` alla volta; per questo motivo alla prima iterazione il *diamond step* viene eseguito solo una volta poiché `matrixSize`, come già detto, inizialmente ha come valore l'ultimo indice di riga. Una volta eseguito questo passaggio per tutti i quadrati vengono eseguiti gli *square steps*. Per essi la definizione dei cicli è un po' più complessa poiché, a differenza dei *diamond steps*, il cui valore centrale viene calcolato aggiungendo $matrixSize/2$ agli indici di riga e di colonna, gli *square steps* devono inizializzare elementi su righe e colonne diverse. In Figura 10 viene utilizzato un esempio rappresentante una matrice $9x9$: all'inizio `matrixSize` è 8, di conseguenza il valore inizializzato durante il primo *diamond step* è:

Equazione 2 - indice del primo elemento inizializzato con diamond step in una matrice 9x9

$$\left(0 + \frac{matrixSize}{2}, 0 + \frac{matrixSize}{2}\right) = (4, 4)$$

Per il primo *square step*, invece, gli elementi da inizializzare sono $(0,4), (4,0), (4,8), (8,4)$, colorati in giallo in Figura 10. Da ciò si può notare che per le righe il cui indice è multiplo di `matrixSize` verrà inizializzato un solo elemento, nella colonna centrale, altrimenti i valori da calcolare saranno due, agli estremi della riga. Per questo motivo, nel Frammento 11, il ciclo che scorre le righe per lo *square step* parte da indice 0 fino all'ultimo, 2^n, incrementando di $matrixSize/2$, poi, in base al valore dell'indice corrente della riga viene definito il ciclo che scorre le colonne: se tale indice è multiplo di `matrixSize` la colonna parte da $matrixSize/2$ fino al penultimo indice di colonna, altrimenti parte da 0 fino all'ultimo; in entrambi i casi il ciclo incrementa la colonna di `matrixSize`. Al termine dello *square step* il metodo `Diamond-`

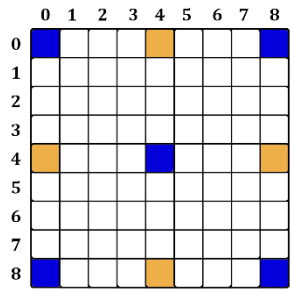

Figura 10 - Elementi da inizializzare con il primo square step in una matrice 9x9

`Square(…)` richiama se stesso dimezzando `matrixSize`, per far sì che ai passi successivi la dimensione del lato della matrice su cui eseguire i vari passaggi sia sempre più piccola, fino a raggiungere il valore 1, quando tutti gli elementi saranno stati creati; anche `maxValue` viene dimezzato, poiché il valore casuale deve essere preso da intervalli sempre più piccoli. Osservando sempre il Frammento 11 occorre porre attenzione ai vari valori finali dei cicli con cui vengono eseguiti *diamond steps* e *square steps*: per i *diamond steps*, infatti, l'ultimo indice di una riga (e di una colonna) preso in considerazione deve essere minore dell'ultimo elemento della riga stessa (o della colonna). Questo è dovuto al fatto che, essendo la matrice di lato dispari ed incrementando la riga (o la colonna) di $matrixSize/2$, l'ultimo indice della riga (o colonna) risulterebbe multiplo di tale quantità, quindi sarebbe uno degli indici usati per eseguire il *diamond step*. Se fosse così, nell'esempio precedente di Figura 10, alla prima iterazione, oltre all'elemento $(4,4)$, verrebbe calcolato il valore anche per gli elementi:

Equazione 3 - valori errati ottenuti se nei cicli dei diamond steps vengono considerati anche l'ultimo indice di riga o colonna

$$\left(0+\frac{matrixSize}{2},\ 8+\frac{matrixSize}{2}\right)=(4,12),$$
$$\left(8+\frac{matrixSize}{2},0+\frac{matrixSize}{2}\right)=(12,4),$$
$$\left(8+\frac{matrixSize}{2},8+\frac{matrixSize}{2}\right)=(12,12),$$

che ovviamente non sono elementi validi perché al di fuori del dominio della matrice. Per il ciclo che scorre le colonne per gli *square steps* può essere fatto un discorso analogo quando la riga corrente è multiplo di `matrixSize`; in questo caso, infatti, oltre ai valori $(0,4)$ e $(8,4)$ verrebbero inizializzati anche $(0,8)$ e $(8,8)$ che tuttavia sono due degli elementi iniziali, perciò non necessitano di essere modificati. Per i cicli di scorrimento delle righe e per quello delle colonne in caso la riga non sia multiplo di `matrixSize`, invece, è necessario raggiungere anche l'ultimo indice, quindi l'ultimo valore dell'indice deve essere minore della dimensione del lato della matrice.

Per eseguire i *diamond steps* viene utilizzato il metodo `DiamondStep (int row, int column, int adding, int maxValue)`. Tale metodo riceve come parametri la riga (`row`) e la colonna (`column`) definite dai cicli in `DiamondSquare(…)`, un valore (`adding`) da aggiungere a tali indici per trovare l'elemento corretto e il valore massimo (`maxValue`) usato per il calcolo del valore casuale; il

parametro `adding` è $matrixSize/2$ per ogni iterazione di `DiamondSquare(…)`, poiché gli elementi ricercati sono quelli centrali al quadrato, quindi con distanza uguale alla metà del lato di quest'ultimo. Per inizializzare il valore dell'elemento richiesto viene fatta una media degli angoli e viene aggiunto un valore casuale; per evitare tuttavia che il risultato si discosti troppo dalla media degli altri elementi, invece di sommare semplicemente il valore casuale, esso è stato sommato agli altri valori prima di eseguire la media ed in seguito la somma così ottenuta è stata divisa per quattro, ovvero il numero degli angoli, come mostrato nel Frammento 12.

```
…
int value = this->image[row*this->size + column]
    + this->image[row*this->size + target_c +
    adding] + this->image[(target_r + adding)
    * this->size + column] +
    this->image[(target_r + adding) * this->size
    + target_c + adding];
value += randomValue;
value /= 4;
…
```

Frammento 12 - codice del metodo DiamondStep

La variabile `randomValue` presente nel frammento è stata precedentemente definita e inizializzata come valore casuale nell'intervallo $-maxValue/2$ e $maxValue/2$, oppure tra -1 e 1: essendo il valore massimo iniziale 256, infatti, dato che tale valore deve essere dimezzato ad ogni iterazione dell'algoritmo, con una matrice di grandi dimensioni si rischia che il numero di iterazioni superi il numero di divisioni intere possibili su 256. Questo numero, infatti, corrisponde a 2^8, quindi è possibile dividerlo per due solamente 8 volte: se si superasse questo numero di divisioni otterremmo $1/2$ che, considerando dividendo e divisore come interi, come risultato dà 0; `maxValue`, tuttavia, viene utilizzato per calcolare il minimo e il massimo dell'intervallo possibile per il valore casuale, calcolato come `randomValue = min + rand() % (max - min) = rand()%0`, che corrisponde al resto della divisione di `rand()` per 0, che ovviamente non è possibile. Per evitare ciò, prima di inizializzare `min` e `max` viene controllato che $maxValue/2$ sia maggiore di 0, in caso contrario vengono usati i valori -1 e 1.

L'esecuzione degli *square steps*, invece, è svolta dal metodo `SquareStep(int row, int column, int adding, int maxValue)`, dove i parametri sono analoghi a quelli del metodo `DiamondStep(…)`; in questa nuova funzione, tuttavia, gli indici di riga e di colonna passati corrispondono già all'elemento che deve essere inizializzato, senza dover aggiungere `adding` come invece era necessario per l'altro metodo. Per il calcolo dell'intervallo del valore casuale valgono le considerazioni fatte in precedenza e anche il calcolo della media è stato svolto nello stesso modo; esiste tuttavia una differenza nel numero di angoli considerati, poiché, per elementi ai lati della matrice gli angoli effettivi sono solo tre, dato che uno cade al di fuori del dominio. In questo caso, dunque, verrà controllata la presenza di ogni angolo e solo quelli esistenti saranno sommati al valore casuale e poi divisi per il loro numero effettivo.

3.2.2. Versione parallela con OpenMP

Partendo dalla versione seriale è stata creata una versione parallela con OpenMP, contenuta nella classe `OpenMPDiamondSquare`. A differenza di `SerialDiamondSquare` che richiamava solamente il costruttore della classe base, per questa nuova classe è necessario che nel costruttore venga definito il numero di *threads* da utilizzare per l'esecuzione dell'algoritmo tramite `omp_set_num_threads(num)`, dove `num` corrisponde al numero di *threads* che l'utente ha specificato nell'interfaccia grafica. Per poter parallelizzare questo algoritmo è necessario analizzare le diverse iterazioni del programma, riportante in pseudo-codice nel seguente frammento.

```
funzione ExecuteDiamondSquare():intero a 8 bit
    dimensione = lato-matrice - 1
    finché dimensione > 1 ripeti
          finché riga < ultima-riga ripeti
                finché colonna < ultima-colonna
                      ripeti
                      esegui DiamondStep()
                fine finché
          fine finché
          finché riga < lato-matrice ripeti
                se riga%matrixSize = 0 allora
                      finché colonna < ultima-
                            colonna ripeti
```

```
                        esegui SquareStep()
                  fine finché
            altrimenti
                  finché colonna < lato-matrice
                        ripeti
                        esegui SquareStep()
                  fine finché
            fine se
      fine finché
   dimensione = dimensione/2
   fine finché
fine funzione
```

Frammento 13 - pseudocodice dell'esecuzione dell'algoritmo Diamond-Square

Il primo ciclo che si incontra è quello che controlla che la dimensione della matrice su cui vengono svolti i calcoli sia sempre maggiore di 1, che nella versione seriale era dato dalle iterazioni ricorsive della funzione `DiamondSquare(…)`. Se venisse parallelizzato tale ciclo, tuttavia, i diversi *threads* lanciati potrebbero lavorare utilizzando elementi non ancora inizializzati, poiché ogni iterazione di *diamond step* e di *square step* si basa sugli elementi precedentemente inizializzati per calcolare il valore dell'elemento corrente. Da questa considerazione si è deciso di lasciare `DiamondSquare(…)` come metodo ricorsivo, mentre se fosse stato possibile parallelizzarlo sarebbe stato necessario trasformare le diverse iterazioni ricorsive in un unico ciclo for. I *diamond steps* e gli *square steps* a loro volta non possono essere eseguiti contemporaneamente perché gli elementi calcolati dal primo vengono usati come angoli dal secondo; tuttavia tutti i valori calcolati dal singolo passaggio sono indipendenti tra loro. Durante un *diamond step* o uno *square step*, infatti, gli angoli per i vari elementi da inizializzare vengono presi tra quelli calcolati nei passaggi precedenti, ma non tra quelli dello stesso passaggio: questo fa sì che i cicli che eseguono i *diamond steps* possono essere parallelizzati, così come quelli per gli *square steps*. Tale parallelizzazione può essere svolta inserendo prima dei cicli la direttiva OpenMP `#pragma omp parallel for`, benché in questo modo verrebbe creato una regione parallela per ogni ciclo parallelizzato. Di per sé questa scelta non sarebbe scorretta, tuttavia la creazione di più *pool* di *threads* può risultare abbastanza onerosa e in questo caso anche inutile dato che i diversi cicli devono essere eseguiti sequenzialmente tra loro: al termine

di uno di essi i suoi *threads* possono essere utilizzati per eseguirne un altro, poiché grazie alla barriera implicita alla fine del for tutti i *threads* avranno già concluso le loro computazioni sul ciclo precedente. Una soluzione migliore consiste nell'utilizzare la direttiva `#pragma omp parallel` all'interno del costrutto condizionale *if* del metodo ricorsivo in modo tale che tutto ciò che è dentro ad esso, tranne la chiamata ricorsiva, venga considerato all'interno della regione parallela, e far precedere i cicli da parallelizzare da `#pragma omp for`. In questo modo ogni *thread* creato eseguirà le istruzioni contenute all'interno della regione parallela e una volta raggiunti i for spartirà con gli altri le varie iterazioni, eseguendo quindi solo quelle che gli sono state assegnate. Ogni chiamata a tale metodo, però, creerà un proprio *pool* di *threads*, quindi la soluzione descritta può essere ulteriormente migliorata creando la regione parallela prima della chiamata a `DiamondSquare(…)` contenuta in `ExecuteDiamondSquare(…)`, inserendo la direttiva OpenMP come mostrato nel Frammento 14.

```
uint8 * OpenMPDiamondSquare::ExecuteDiamondSquare()
{
    …
#pragma omp parallel
    {
          this->DiamondSquare(last, MAX);
    }
    return this->image;
}
```

Frammento 14 - frammento di codice della funzione ExecuteDiamondSquare per OpenMP

Durante la fase di parallelizzazione tramite OpenMP è necessario porre attenzione anche alla creazione dei valori casuali: generalmente, infatti, le funzioni `srand()` e `rand()` utilizzate per generare valori pseudo-casuali non sono *thread-safe*, quindi se usate in codici *multi-threaded* provocherebbero *race-conditions* perché accedono ad informazioni condivise. Visual Studio, tuttavia, mette a disposizione librerie Runtime C [24] basate sul metodo di programmazione TLS (thread local storage), grazie al quale per le variabili che normalmente sarebbero condivise, vengono create copie private per ogni *thread* in esecuzione. Le funzioni `rand()` e `srand()`, quindi, possono essere utilizzate senza problemi; tuttavia, mentre normalmente è sufficiente una sola invocazione di quest'ultima funzione citata, a causa del TLS deve

essere invocata una volta per ogni *thread* mandato in esecuzione [25]; per poter generare sequenze diverse di numeri pseudo-casuali, inoltre, `srand()` deve essere inizializzato sempre con un valore diverso. Di solito è sufficiente utilizzare la funzione `time(dest)` [26], che restituisce l'ora corrente calcolata partendo dal 1° gennaio 1970, perciò se venisse invocata nello stesso istante da più *threads* restituirebbe lo stesso valore per ognuno di essi, inizializzando nello stesso modo tutte le diverse `srand()`. Per evitare questo problema in `OpenMP-DiamondSquare` le diverse sequenze di numeri pseudo-casuali sono state generate come mostrato nel Frammento 15.

```
…
unsigned int seed =
    (unsigned)time(0)*(omp_get_thread_num() + 1);
srand(seed);
…
```

Frammento 15 - porzione di codice per la generazione di sequenze di numeri pseudocasuali

Il seme con cui generare i valori casuali utilizza come base l'ora corrente di `time(…)`, poi tale valore viene moltiplicato per il numero che identifica il *thread* corrente, sommato a uno perché l'indicizzazione dei *threads* parte da 0. In questo modo ogni *thread* avrà la propria sequenza di numeri da cui `rand()` recupererà il valore casuale da utilizzare per l'inizializzazione degli elementi della matrice.

3.2.3. Versione parallela con CUDA

La classe utilizzata per la versione CUDA è `CudaDiamondSquare`; tuttavia la vera e propria implementazione è contenuta nella libreria statica DiamondSquareCuda, al cui interno sono presenti la funzione che esegue l'algoritmo e i kernel che eseguono i calcoli sulla GPU. All'interno di questa libreria è definita la classe `CudaAlgorithm`, contenente il metodo statico `CudaDiamondSquare(uint8_t* matrix, int matrixSize, int randomValue)`, che nell'omonima classe del progetto verrà richiamato all'interno di `DiamondSquare(…)`. I parametri passati al metodo `CudaDiamondSquare(…)` sono la matrice che si vuole riempire, opportunamente allocata precedentemente e con gli angoli già inizializzati in `ExecuteDiamondSquare()`, la dimensione del lato di tale matrice e il valore massimo iniziale usato per i valori casuali. Partendo da questi parametri in entrata, nel metodo verrà allocato il

puntatore alla memoria globale della GPU che conterrà i valori della matrice calcolati dai kernel, poiché non è possibile lavorare direttamente sulla memoria dell'*host* dal *device*. Oltre alla classe `CudaAlgorithm` la libreria contiene due kernel, uno per l'esecuzione dei *diamond steps* e uno per quella degli *square steps*: nel primo caso il kernel è `DiamondStep(uint8_t* matrix, unsigned *random, int currentSize, int matrixSize, int randValue)`, dove `matrix` è il puntatore al *device* per la matrice e `random` il puntatore contenente i valori casuali, la cui generazione verrà spiegata in seguito. Il parametro `currentSize` indica la dimensione del lato della porzione di matrice per cui si sta per calcolare il valore centrale, mentre `matrixSize` è la dimensione totale del lato della matrice; infine `randValue` è il valore rappresentante gli estremi dell'intervallo da cui verrà estratto il numero casuale. Il secondo kernel, invece, è `SquareStep(uint8_t* matrix, unsigned* random, int currentSize, int matrixSize, int maxRowThread, int maxColThread, int randValue)`: i primi quattro parametri e l'ultimo sono analoghi a quelli del kernel appena descritto, mentre `maxRowThread` e `maxColThread` indicano rispettivamente la lunghezza massima della riga e della colonna della matrice di *thread* mandata in esecuzione sulla GPU.

In riferimento al Frammento 13 (a pagina 32), il primo ciclo presente nello pseudo-codice è stato implementato all'interno del metodo `CudaDiamondSquare(…)` tramite costrutto `while`, poiché, come già detto, questo ciclo non può essere parallelizzato. I cicli successivi, invece, sono stati parallelizzati utilizzando il parallelismo offerto dai *threads* della GPU. Per poter eseguire tale parallelismo è necessario specificare dall'*host* quanti *threads* vogliamo mandare in esecuzione contemporaneamente; una prima possibilità è quella di generare tanti *threads* quanti sono gli elementi della matrice, quindi $(2^n + 1)x(2^n + 1)$ *threads* per eseguire ogni kernel, tuttavia, per ogni iterazione di *diamond steps* o di *square steps* gli elementi della matrice realmente modificati sono un numero molto piccolo, quindi la maggior parte dei *threads* creati in questo modo resterebbero inutilizzati. Per evitare ciò è stato studiato il comportamento di ogni passaggio, in modo da poter capire quanti elementi vengono realmente inizializzati ad ogni iterazione dell'algoritmo. Prendendo come esempio l'esecuzione mostrata in Figura 5 a pagina 21, nella seguente Figura 11 vengono mostrati i vari elementi inizializzati tramite le varie iterazioni di *diamond steps*. Le celle colorate di blu indicano i valori

precedentemente inizializzati, mentre quelle gialle sono gli elementi che devono essere calcolati nel passo corrente. Per semplicità, essendo uno studio sull'implementazione dei *diamond steps*, vengono riportati solamente i passi di questo tipo tralasciando i vari *square steps* eseguiti tra uno e l'altro passaggio e quelli successivi che portano al completamento della matrice.

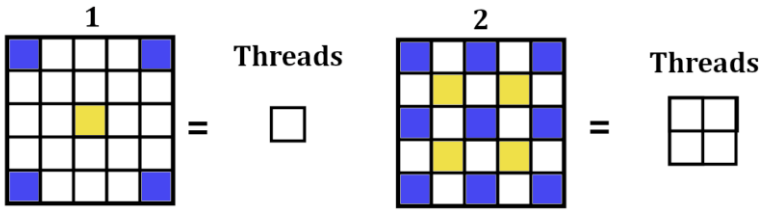

Figura 11 - Rappresentazione grafica dei thread utilizzati per l'esecuzione di Diamond Steps

Come di nota, al primo *diamond step* è sufficiente un solo *thread* in esecuzione sul *device* per calcolare l'unico elemento da inizializzare, mentre per il secondo i valori da calcolare sono quattro, quindi è stato scelto di lanciare un blocco in due dimensioni contenente i quattro *threads* da utilizzare. Per questa matrice $5x5$ non sono eseguiti altri *diamond steps* oltre a quelli mostrati in figura, comunque è possibile generalizzare il discorso anche per qualsiasi altra dimensione scelta. Da questo esempio, infatti, si osserva che, data la matrice $5x5 = (2^2 + 1)x(2^2 + 1)$, alla prima iterazione è necessario utilizzare un blocco $2^0x2^0 = 1$, mentre per la seconda esso avrà dimensioni $2^1x2^1 = 2x2$, quindi, generalizzando:

Equazione 4 - calcolo del numero dei vari threads per l'esecuzione dei vari Diamond Steps

$$Data\ la\ matrice\ (2^n + 1)x(2^n + 1):$$
$$Passo\ 1 \rightarrow 2^0x2^0 = 1\ thread$$
$$Passo\ 2 \rightarrow\ 2^1x2^1 = 4\ threads$$
$$\dots$$
$$Passo\ n \rightarrow 2^{n-1}x2^{n-1}\ threads$$

Una volta creati blocchi con queste dimensioni, ogni *thread* avrà un proprio indice (x, y), dove:

```
x = blockIdx.x * blockDim.x + threadIdx.x
y = blockIdx.y * blockDim.y + threadIdx.y
```

Frammento 16 - indice univoco in un thread all'interno di un blocco in due dimensioni

`blockIdx` indica le coordinate (x, y) all'interno della griglia di blocchi, anch'essa in due dimensioni, mentre `threadIdx` raggruppa le

coordinate per il *thread* all'interno del blocco stesso; `blockDim.x`, invece indica la lunghezza delle righe della griglia e `blockDim.y` la dimensione delle sue colonne. Ogni *thread*, in base al suo indice, sarà così in grado di calcolare un elemento della matrice, ma, mentre nel caso in cui i *threads* siano lo stesso numero degli elementi i loro indici equivalgono alle effettive coordinate dell'elemento da inizializzare, creando solamente un numero ridotto di *threads*, per poter raggiungere la posizione esatta all'interno della matrice è necessario che le coordinate di questi ultimi siano traslate. Riferendosi sempre all'esempio di Figura 11, infatti, per calcolare l'elemento (2,2) dovrà essere usato il *thread* (0,0), mentre per i valori in (1,1), (1,3), (3,1) e (3,3) verranno utilizzati rispettivamente i *threads* (0,0), (0,1), (1,0) e (1,1). Da ciò si può osservare che la traslazione necessaria è la seguente:

Equazione 5 - calcolo delle coordinate dell'elemento traslando quelle dei threads

$$x_{elemento} = x_{thread} * currentSize + \frac{currentSize}{2}$$

$$y_{elemento} = y_{thread} * currentSize + \frac{currentSize}{2}$$

$$con\ currentSize = \frac{matrixSize - 1}{2^{k-1}}, dove\ k = passo\ corrente$$

Nel primo passaggio mostrato in figura, infatti, il *thread* da utilizzare ha coordinate (0,0), $currentSize = \frac{5-1}{2^{1-1}} = 4$, mentre l'elemento è in posizione $\left(0 * 4 + \frac{4}{2}, 0 * 4 + \frac{4}{2}\right) = (2,2)$; nel secondo, invece:

Equazione 6 - coordinate degli elementi che devono essere inizializzati al secondo Diamond Step dell'esempio

$$currentSize = \frac{5-1}{2^{2-1}} = 2$$

$$thread\ (0,0) \rightarrow elemento\ (0 * 2 + 1, 0 * 2 + 1) = (1,1)$$
$$thread\ (0,1) \rightarrow elemento\ (0 * 2 + 1, 1 * 2 + 1) = (1,3)$$
$$thread\ (1,0) \rightarrow elemento\ (1 * 2 + 1, 0 * 2 + 1) = (3,1)$$
$$thread\ (1,1) \rightarrow elemento\ (1 * 2 + 1, 1 * 2 + 1) = (3,3)$$

Poiché il numero di *threads* per blocco è ridotto, inoltre, è possibile che all'aumentare della dimensione della matrice sia necessario un numero di *threads* maggiore della dimensione massima del blocco, quindi deve essere predisposto un controllo per far sì che, in caso di necessità, venga creata una griglia contenente più di un blocco, in modo da poter

suddividere tra essi i vari *threads* da mandare in esecuzione. Dato che il valore massimo di *threads* per blocco è stato impostato a 512, la dimensione massima di un blocco non può essere superiore a $16x16 = 256$, poiché la grandezza successiva, $32x32$, conterrebbe 1024 *threads*. La dimensione delle righe e delle colonne di una griglia, quindi, sarà:

Equazione 7 - possibili dimensioni di una griglia di blocchi

$$\begin{cases} \frac{2^k}{16} \; se \; k > 4 \\ 1 \; altrimenti \end{cases}, con \; k = 0,1, \dots, n-1$$

mentre le dimensioni di un blocco sono:

Equazione 8 - possibili dimensioni di un blocco di threads

$$\begin{cases} 2^k \quad se \; k \leq 4 \\ 16 \; altrimenti \end{cases}$$

Poiché la dimensione della griglia viene calcolata con una divisione tra interi, il risultato di essa, nel caso il dividendo non sia multiplo del divisore, viene sempre approssimato per difetto: il resto verrebbe quindi scartato, perciò il numero effettivo di *threads* mandati in esecuzione risulterebbe minore rispetto al numero di quelli richiesti. In questo caso, tuttavia, questo problema non si pone perché $16 = 2^4$ e 2^k è sempre suo multiplo per $k > 4$.

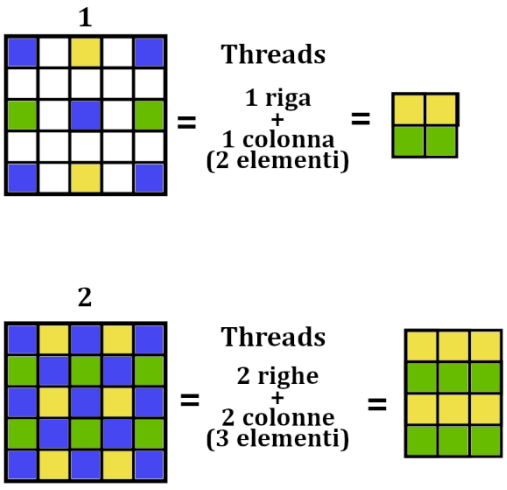

Figura 12 - Rappresentazione grafica dei thread utilizzati per l'esecuzione di Square Steps

Per gli *square steps* il discorso è più complesso, come si nota nella figura precedente (Figura 12), in cui, come per l'immagine precedente, vengono riportati solo i passaggi in analisi escludendo la visualizzazione dei *diamond steps* eseguiti prima e dopo ad essi. A differenza degli elementi inizializzati con i *diamond steps*, che sono posizionati all'interno della matrice come angoli di quadrati, gli elementi ottenuti tramite *square steps* sono divisi in ugual numero su righe e su colonne sfalsate. Nell'esempio, infatti, al primo *square step* due elementi sono disposti sulla stessa riga, mentre altri due sulla stessa colonna ma su righe diverse rispetto ai primi; nel secondo passaggio, invece, sei elementi sono disposti su due righe, tre per ognuna, e altri sei su due colonne. Essendo così disposti il modo più semplice per la creazione dei *threads* è utilizzare un blocco con tante righe quante sono le righe e le colonne occupate dagli elementi da inizializzare, tutte lunghe il numero di elementi su ogni riga o colonna. Per comprendere meglio il ragionamento si può osservare la disposizione degli elementi in Figura 12: le celle da inizializzare disposte sulle colonne sono state colorate in giallo, mentre quelle nelle righe sono verdi; a fianco è riportato il blocco di *threads* utilizzato per il calcolo dei valori di tali celle ed è evidente che ogni riga del blocco ha il compito di calcolare gli elementi disposti su una riga o una colonna della matrice, mentre il numero di colonne corrisponde a quanti elementi da inizializzare sono contenuti in esse. Da queste considerazioni si può notare che al primo passo serve un blocco $2x2$, mentre per il secondo uno da $2^2x(2^{2-1}+1) = 4x3$; quindi, in generale si ha:

Equazione 9 - calcolo del numero dei vari threads per l'esecuzione dei vari Square Steps

$$Data\ una\ matrice\ (2^n+1)x(2^n+1):$$
$$Passo\ 1 \rightarrow 2^1x(2^{1-1}+1) = 2x2\ threads$$
$$Passo\ 2 \rightarrow 2^2x(2^{2-1}+1) = 4x3\ threads$$
$$\dots$$
$$Passo\ n \rightarrow 2^nx(2^{n-1}+1)\ threads$$

Dato che alle righe dei blocchi non corrispondono solo righe della matrice ma anche delle sue colonne, anche la traslazione delle coordinate dei *threads* per trovare la posizione degli elementi da inizializzare risulta più complessa. Tale traslazione, infatti, deve essere eseguita in due modi diversi nel caso la cella da inizializzare sia su una

riga o su una colonna, e questa distinzione viene fatta controllando il valore della coordinata y del *thread* come descritto dall'Equazione 10.

Equazione 10 - calcolo delle coordinate dell'elemento tralsando quelle dei threads

$$x_{elemento} = \begin{cases} x_{thread} * currentSize \; se \; y_{thread} \; \% \; 2 = 0 \\ y_{thread} * \dfrac{currentSize}{2} \quad se \; y_{thread} \; \% \; 2 \neq 0 \end{cases}$$

$$y_{elemento} = \begin{cases} y_{thread} * \dfrac{currentSize}{2} + \dfrac{currentSize}{2} \\ \qquad se \; y_{thread} \; \% \; 2 = 0 \\ x_{thread} * currentSize \; se \; y_{thread} \; \% \; 2 \neq 0 \end{cases}$$

Il valore di `currentSize` è, come per i *diamond steps*, la dimensione della matrice dimezzata tante volte quanto il numero di passi svolti (formula riportata anche nell'Equazione 5 a pagina 37), mentre la dimostrazione di queste formule può essere fatta nel modo seguente considerando gli elementi inizializzati durante il secondo *square step*, riportato nell'esempio di Figura 12 a pagina 38:

Equazione 11 - coordinate degli elementi che devono essere inizializzati durante il secondo Square Step dell'esempio

$$matrixSize = 5, passo\ 2 \rightarrow currentSize = \frac{5-1}{2^{2-1}} = \frac{4}{2} = 2$$
$$riga\ 1\ del\ blocco \rightarrow y_{thread} = 0 \rightarrow y_{thread}\ \%\ 2 = 0$$
$$thread\ (0{,}0) \rightarrow (0 * 2, 0 * 1 + 1) = (0{,}1)$$
$$thread\ (1{,}0) \rightarrow (1 * 2, 0 * 1 + 1) = (2{,}1)$$
$$thread\ (2{,}0) \rightarrow (2 * 2, 0 * 1 + 1) = (4{,}1)$$
$$riga\ 2\ del\ blocco \rightarrow y_{thread} = 1 \rightarrow y_{thread}\ \%\ 2 \neq 0$$
$$thread\ (0{,}1) \rightarrow (1 * 1, 0 * 2) = (1{,}0)$$
$$thread\ (1{,}1) \rightarrow (1 * 1, 1 * 2) = (1{,}2)$$
$$thread\ (2{,}1) \rightarrow (1 * 1, 2 * 2) = (1{,}4)$$
$$riga\ 3\ del\ blocco \rightarrow y_{thread} = 2 \rightarrow y_{thread}\ \%\ 2 = 0$$
$$thread\ (0{,}2) \rightarrow (0 * 2, 2 * 1 + 1) = (0{,}3)$$
$$thread\ (1{,}2) \rightarrow (1 * 2, 2 * 1 + 1) = (2{,}3)$$
$$thread\ (2{,}2) \rightarrow (2 * 2, 2 * 1 + 1) = (4{,}3)$$
$$riga\ 4\ del\ blocco \rightarrow y_{thread} = 3 \rightarrow y_{thread}\ \%\ 2 \neq 0$$
$$thread\ (0{,}3) \rightarrow (3 * 1, 0 * 2) = (3{,}0)$$
$$thread\ (1{,}3) \rightarrow (3 * 1, 1 * 2) = (3{,}2)$$
$$thread\ (2{,}3) \rightarrow (3 * 1, 2 * 2) = (3{,}4)$$

L’implementazione delle formule dell’Equazione 10 può essere eseguita seguendo lo pseudo-codice riportato nel Frammento 17.

```
half = currentSize/2
se y_thread % 2 = 0 allora
    x_elemento = x_thread * currentSize
    y_elemento = y_thread * half + half
altrimenti
    x_elemento = y_thread * half
    y_elemento = x_thread * currentSize
fine se
```

Frammento 17 - pseudo-codice per il calcolo delle coordinate degli elementi durante gli Square steps

Da tale frammento si nota la presenza di programmazione condizionale per cui, in base al valore di una variabile vengono eseguite alcune istruzioni invece di altre; di per sé questo non sarebbe un problema, tuttavia i vari *threads* di uno stesso *warp* mandati in esecuzione contemporaneamente devono eseguire tutti la stessa istruzione. In caso di condizioni è molto probabile che le istruzioni da eseguire divergano, lasciando alcuni *threads* ad eseguire quelle per quando la condizione è vera ed altri quelle per quando essa è falsa: i diversi blocchi condizionali verranno, quindi, eseguiti sequenzialmente, lanciando prima i *threads* incaricati dell’esecuzione della condizione vera, poi quelli per la condizione falsa [27]. Anche se ciò è vero solo per *threads* di uno stesso *warp*, mentre *threads* di *warps* diversi possono essere eseguiti indipendentemente, in caso di programmazione divergente l’esecuzione potrebbe essere più lenta, perciò è stato trovato un altro modo per implementare lo pseudo-codice del Frammento 17.

```
…
int elemX = x * currentSize*(y % 2 == 0) +
    y * half*(y % 2 != 0);
int elemY = (y*half + half)*(y % 2 == 0) +
    x * currentSize*(y % 2 != 0);
…
```

Frammento 18 - frammento di codice estrapolato dal kernel che esegue gli Square Steps

Il Frammento 18, infatti, mostra come vengono calcolate le coordinate degli elementi all’interno del kernel `SquareStep(…)`: la variabile `elemX` indica la coordinata x e viene calcolata sommando le due formule possibili moltiplicate ognuna per la condizione a cui si

riferiscono, lo stesso vale per la variabile `elemY`, indicante la coordinata y. Questo metodo è possibile perché ogni condizione booleana può essere considerata come un intero di valore 0 o 1, perciò, moltiplicando un numero per un booleano, se quest'ultimo è vero il numero resta invariato perché moltiplicato per 1, se invece è falso il risultato della moltiplicazione è 0. Le variabili `elemX` ed `elemY`, quindi, saranno sempre il risultato di una sola delle due espressioni perché tra le due condizioni con cui vengono moltiplicate solamente una sarà vera, mentre l'altra annullerà l'espressione a cui è moltiplicata. Un'altra differenza rispetto all'esecuzione dei *diamond steps* sta nella suddivisione di diversi *threads* in blocchi: anche in questo caso è necessario creare più blocchi nel caso la quantità di *threads* superi 512 e per restare sotto a questa soglia ogni blocco potrà contenere fino a $2^4 x(2^3 + 1) = 16x9 = 144$ *threads*, perché la dimensione successiva, $32x17 = 544$, supera la dimensione massima. Per la divisione in blocchi, quindi, se la grandezza supera $16x9$ è necessario far sì che il numero di righe venga diviso per 16 e quello delle colonne per 9, tuttavia, come si nota anche dai due esempi sopra riportati, per quanto 32 sia divisibile per 16, 17 non lo è per 9, quindi $17/9 = 1$, dato che è una divisione per numeri interi. Se viene eseguita tale divisione, quindi, otto colonne di *threads* andrebbero perse e parte degli *square steps* non verrebbe eseguita; per evitare questo è necessario modificare la divisione come segue:

Equazione 12 - suddivione delle colonne di un blocco di threads in più blocchi

$$((2^k + 1) + 9 - 1)/9, con\ k > 3$$

In questo modo vengono creati più *threads* di quelli richiesti, ma si evita di perdere il resto della divisione precedente. All'interno del kernel, poi, sarà necessario controllare che il *thread* corrente faccia parte di quelli effettivi utili per l'esecuzione dell'algoritmo, escludendo quelli che invece sono stati creati solo per completare la creazione del blocco.

Per quanto riguarda il calcolo dei valori casuali, in questa versione parallela non è possibile utilizzare la funzione `rand()`, ma CUDA mette a disposizione una libreria per la generazione di numeri casuali chiamata cuRAND [28]. Grazie ad essa, utilizzabile includendo nel codice `curand.h`, è possibile creare generatori di valori casuali di tipo `curandGenerator_t`, inizializzati tramite la funzione `curand-`

`CreateGenerator()` in cui è possibile definire che tipo di generatore si vuole usare. I tipi messi a disposizione dalla libreria sono nove e si dividono in due categorie: una contenente le cinque tipologie che generano numeri pseudo-casuali, l'altra per i restanti quattro in grado di generare valori quasi-casuali. Quest'ultimo concetto indica i valori di una sequenza a bassa discrepanza, ovvero una sequenza i cui punti sono distribuiti più equamente all'interno di un dato volume rispetto a una sequenza di valori pseudo-casuali [29]. Nella libreria scritta per questa tesi i generatori utilizzati sono stati creati utilizzando come tipo quello indicato da `CURAND_RNG_PSEUDO_DEFAULT`, che indica il tipo pseudo-casuale `CURAND_RNG_PSEUDO_XORWOW`: esso crea i valori pseudo-casuali utilizzando l'algoritmo XORWOW, il quale, per calcolare i numeri successivi della sequenza esegue operazioni di XOR tra un numero e il risultato di uno SHIFT bit a bit dello stesso [30]. Una volta creato il generatore può essere utilizzato dalla funzione `curandGenerate(generator, output, size)` per creare i valori pseudo-casuali di cui si ha necessità: il parametro `generator` indica il generatore appena creato, `output` è il puntatore ai valori creati e `size` rappresenta il numero di elementi che dovranno essere contenuti in `output`; per il tipo di generatore scelto i valori pseudo-casuali così creati saranno interi senza segno a 32 bit. Nel metodo `CudaDiamondSquare(…)` viene creato un generatore prima di ogni *diamond step* o *square step*, salvando i vari valori pseudo-casuali nel puntatore al *device* `d_random`; esso viene allocato prima di ogni passaggio dandogli come dimensione il numero di *threads* lanciati, in modo che per ogni passo venga creato il numero corretto di valori da utilizzare. Al termine di ogni *diamond step* o *square step* la memoria allocata per il puntatore viene liberata con `cudaFree(d_random)` e il generatore viene distrutto tramite `curandDestroyGenerator (generator)` per far sì che entrambi siano riutilizzabili per il passo successivo.

3.3. Analisi delle prestazioni

Una volta create le diverse versioni parallele dell'algoritmo è stato possibile analizzare le prestazioni di tali implementazioni. I calcoli che verranno presentati vengono fatti sui tempi raccolti eseguendo il programma su un computer dotato di una CPU Intel® Core™ i5-8600K[2] e una scheda grafica NVIDIA GeForce 1060 6GB[3]. La CPU adottata è composta da sei Core fisici e non fa uso di Hyper-Threading, ovvero non sfrutta il particolare tipo di implementazione *multithreading* di Intel con cui ogni core fisico viene considerato come due core virtuali. L'utilizzo delle versioni parallele inizia ad essere indicativo superata la dimensione $(2^8 + 1)x(2^8 + 1) = 257x257$, poiché, come riporta la Tabella 1, con una matrice di questa grandezza i tempi sono talmente bassi che utilizzando OpenMP non si nota la differenza tra l'esecuzione con un certo numero di *threads* e quella con un altro.

	p = 1	p = 2	p = 3	p = 4	p = 5	p = 6	p = 7	p = 8	p = 9	p = 10
	0,002	0,002	0,001	0	0	0,002	0,002	0,002	0,002	0,001
	0,001	0	0	0,001	0,001	0	0,001	0,001	0	0,001
	0,002	0,001	0,001	0	0,001	0,001	0,001	0,001	0,001	0
	0,002	0,001	0	0,001	0	0,001	0	0	0	0,001
	0,002	0,001	0,001	0,001	0,001	0,001	0,001	0	0,001	0
MEDIA	**0,0018**	**0,001**	**0,0006**	**0,0006**	**0,0006**	**0,001**	**0,001**	**0,0008**	**0,0008**	**0,0006**

Tabella 1 - Tempi calcolati per una matrice 257x257 utilizzando OpenMP

In Tabella 2, invece, vengono riportati i tempi ottenuti per una matrice di dimensione massima, $(2^{13} + 1)x(2^{13} + 1) = 8193x8193$, e con essi viene fatta anche l'analisi dello *speedup* della versione OpenMP.

[2] Le specifiche della CPU sono consultabili sul sito: https://ark.intel.com/content/www/it/it/ark/products/126685/intel-core-i5-8600k-processor-9m-cache-up-to-4-30-ghz.html

[3] Le specifiche della GPU sono consultabili sul sito: https://www.nvidia.com/it-it/geforce/products/10series/geforce-gtx-1060/

	p = 1	p = 2	p = 3	p = 4	p = 5	p = 6	p = 7	p = 8	p = 9	p = 10
	1,741	0,898	0,593	0,452	0,361	0,332	0,481	0,429	0,39	0,387
	1,745	0,877	0,591	0,452	0,358	0,314	0,495	0,431	0,411	0,38
	1,753	0,878	0,593	0,449	0,358	0,316	0,449	0,431	0,415	0,381
	1,744	0,88	0,591	0,451	0,369	0,317	0,408	0,43	0,391	0,363
	1,738	0,88	0,593	0,448	0,366	0,318	0,498	0,424	0,42	0,408
MEDIA	1,7442	0,8826	0,5922	0,4504	0,3624	0,3194	0,4662	0,429	0,4054	0,3838
S(p)	1	1,976207	2,945289	3,872558	4,812914	5,460864	3,741313	4,065734	4,302417	4,544554

Tabella 2 - Tempi calcolati per una matrice 8193x8193 utilizzando OpenMP

In entrambe le tabelle si nota che l'intestazione verticale si riferisce a una variabile p: essa indica il numero di processi, quindi in questo caso di *threads*, utilizzati per eseguire l'algoritmo. Viene usata la variabile p di processo al posto di t di *thread* per evitare di confondere questa grandezza con il tempo che in seguito verrà definito con la lettera t. $S(p)$ riportato in Tabella 2, invece, indica lo *speedup* che verrà mostrato nel seguente Grafico 1. Per comprendere meglio tale grafico è necessario spiegare cosa sia lo *speedup*: esso è il rapporto tra il tempo seriale e quello parallelo, dato dalla formula:

Equazione 13 - formula dello speedup

$$S(p) = \frac{t(1)}{t(p)}$$

dove $t(1)$ indica il tempo di esecuzione ottenuto con un solo processo, quindi il tempo seriale, mentre $t(p)$ è il tempo ottenuto con p processi [7]. Nel caso ideale si ha $S(p) = p$, quando il codice risulta completamente parallelizzabile, tuttavia questo caso è difficile da raggiungere perché è sempre presente una porzione di codice non parallelizzabile, che dovrà comunque essere eseguita sequenzialmente. Come osservato da Gene Amdahl [31], infatti, lo *speedup* massimo raggiungibile dipende dal tempo parallelo dato dalla formula:

Equazione 14 - tempo parallelo ottenuto in presenza di porzioni di codice da eseguire sequenzialmente

$$t(p) = \alpha t(1) + \frac{(1-\alpha)t(1)}{p}$$

in cui il tempo parallelo è composto dal tempo di esecuzione della porzione parallela dato da $\frac{(1-\alpha)t(1)}{p}$, sommato al tempo di esecuzione

della porzione α da eseguire serialmente. Da ciò lo *speedup* può essere definito come:

Equazione 15 - Legge di Amdahl

$$S(p) = \frac{t(1)}{t(p)} = \frac{t(1)}{\alpha t(1) + \frac{(1-\alpha)t(1)}{p}} = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Esso, quindi, anche con infiniti processi, non potrà mai superare la grandezza $1/\alpha$.

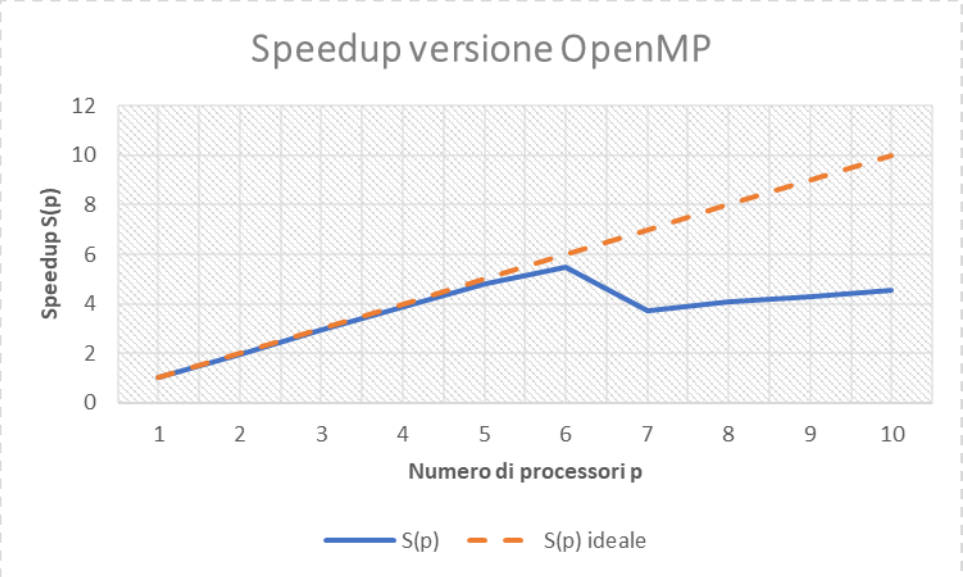

Grafico 1 - Grafico dello speedup della versione OpenMP per una matrice 8193x8193

Nel grafico sopra riportato la linea tratteggiata arancione indica lo *speedup* ideale, mentre quella blu deriva dai dati effettivi contenuti nella precedente Tabella 2 a pagina 45; si può notare che fino a sei processi lo *speedup* prosegue in maniera quasi lineare seguendo la linea ideale, poi i tempi di esecuzione aumentano e lo *speedup* dimininuisce. Sebbene in seguito cresca ancora non riesce a raggiungere il valore del picco a sei processi perché esso dipende dalle capacità dell'hardware su cui viene eseguito l'algoritmo: come già accennato, infatti, la CPU utilizzata è composta da sei core, quindi le sue prestazioni massime si raggiungono sfruttandoli tutti. Aumentando il numero di *threads* di OpenMP e superando la quantità di processori disponibili, ognuno di questi ultimi avrà il compito di eseguire le istruzioni di più *threads*,

quindi i tempi si allungano perché non tutti i *threads* verranno eseguiti contemporaneamente, ma verranno schedulati tra i vari processori disponibili. Oltre allo *speedup*, per questa versione parallela è possibile calcolare anche la *strong scaling efficiency* [7]: essa indica la scalabilità dell'algoritmo mantenendo la dimensione del problema fissa e aumentando il numero di processi con cui eseguirlo e si calcola come:

Equazione 16 - Strong scaling efficiency

$$E(p) = \frac{S(p)}{p}$$

da ciò si può dedurre che, in caso di *speedup* ideale, $S(p) = p$,

Equazione 17 - Strong scaling efficiency ideale

$$E(p) = \frac{p}{p} = 1$$

quindi, dato che tale *speedup* è difficile da raggiungere, l'efficienza reale risulterà essere minore di 1. Nella Tabella 3 vengono riportati i dati utilizzati per la creazione del grafico dell'efficienza.

p	1	2	3	4	5	6	7	8	9	10
S(p)	1	1,976207	2,945289	3,872558	4,812914	5,460864	3,741313	4,065734	4,302417	4,544554
E(p)	1	0,988103	0,981763	0,968139	0,962583	0,910144	0,534473	0,508217	0,478046	0,454455

Tabella 3 - Dati derivanti dall'esecuzione della versione OpenMP su una matrice 8193x8193

La riga p contiene per ogni colonna il numero di processi utilizzati durante l'esecuzione, mentre *S(p)* indica lo *speedup* calcolato con i tempi riportati nella Tabella 2 a pagina 45. *E(p)*, infine, è la riga che contiene l'efficienza calcolata utilizzando la formula dell'Equazione 16 e per cui di seguito viene riportato il grafico (Grafico 2). In esso è presente, come per il grafico dello *speedup*, una linea arancione tratteggiata che indica l'efficienza ideale ottenuta da uno *speedup* lineare, mentre quella blu rappresenta la *strong scaling efficiency* reale riportata in Tabella 3. Essendo lo *speedup* quasi lineare fino a $p = 6$, anche l'efficienza si avvicina molto al suo valore ideale, poi, superato

questo numero di *threads*, l'efficienza inizia a calare perché i tempi di esecuzione aumentano facendo diminuire lo *speedup*.

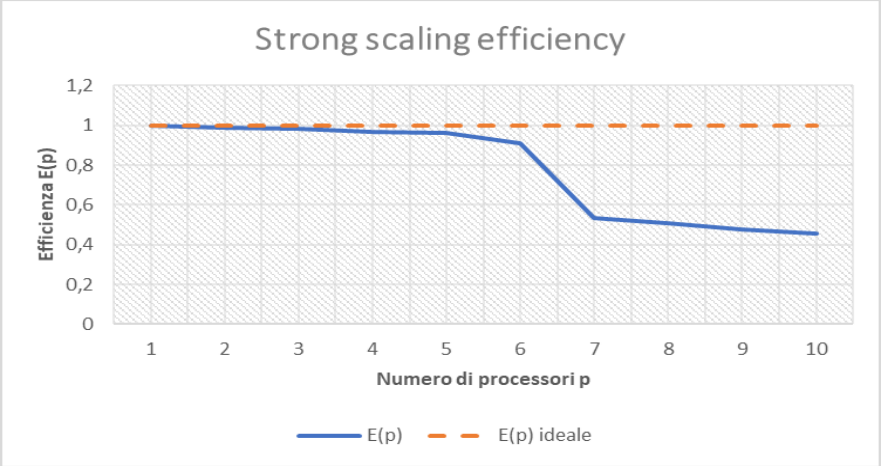

Grafico 2 - Grafico dell'efficienza della versione OpenMP per una matrice 8193x8193

Per quanto riguarda la versione CUDA, invece, non è possibile calcolare lo *speedup* e la *strong scaling efficiency* perché, come descritto in **3.2.3 Versione parallela con CUDA** (a pagina 34), il numero di *threads* con cui viene lanciato ogni passo dell'algoritmo dipende da un calcolo preciso, quindi non è possibile provare l'esecuzione modificando la loro quantità. Per verificare le prestazioni di questa versione viene fatto un confronto tra i tempi di esecuzione di essa e quelli ottenuti con OpenMP, sia utilizzando un solo *thread* per i tempi dell'esecuzione seriale, che usandone sei per i tempi migliori ottenibili per questa versione. Questi confronti vengono fatti nel Grafico 3 calcolando i tempi medi di esecuzione dell'algoritmo su matrici $(2^x + 1)x(2^x + 1)$ con $x = 9, 10, 11, 12, 13$. Tale grafico è composto da istogrammi che mostrano le varie versioni: quelli di colore blu rappresentano le esecuzioni seriali, quelli arancioni indicano i tempi ottenuti con OpenMP, mentre quelli verdi quelli con CUDA. Dai dati riportati si può vedere che per $x < 12$ la versione CUDA è meno efficiente anche di quella seriale, per $x = 12$ CUDA è migliore della seriale, ma OpenMP è ancora più efficiente, mentre per $x = 13$ produce tempi migliori di entrambe le altre due versioni.

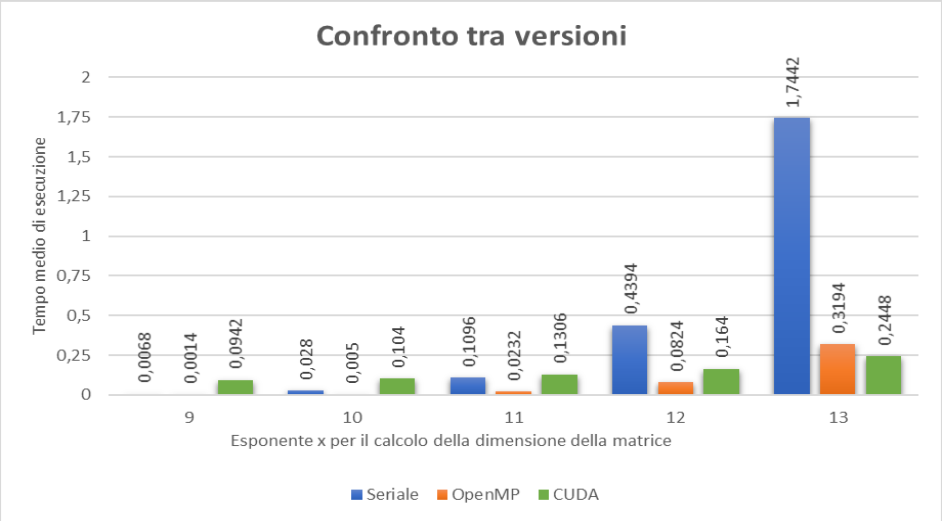

Grafico 3 - Grafico di confronto dei tempi ottenuti con le diverse versioni dell'algoritmo

Questo può essere causato da una programmazione eseguita male perché all'interno del kernel `SquareStep(…)` sono ancora presenti i costrutti condizionali *if* con cui viene controllato che il lato del diamante non venga preso da un indirizzo che non fa parte del dominio della matrice. Per verificare questa ipotesi il codice è stato modificato togliendo i costrutti *if* e sostituendo l'inizializzazione dell'elemento come nel frammento che segue:

```
…
cond = elemX != 0;
value += matrix[(elemX - half * cond) *
    matrixSize + elemY] * cond;
div += cond;
cond = elemX != matrixSize - 1;
value += matrix[(elemX + half * cond) *
    matrixSize + elemY] * cond;
div += cond;
cond = elemY != 0;
value += matrix[elemX * matrixSize +
    elemY - half * cond] * cond;
div += cond;
cond = elemY != matrixSize - 1;
value += matrix[elemX*matrixSize + elemY
    + half * cond] * cond;
div += cond;
value += (minRand + random[x*gridDim.x+y] %
```

```
    (randValue - minRand));
matrix[elemX*matrixSize + elemY] = value / div;
…
```

Frammento 19 - porzione di codice del kernel SquareStep modificato

Nel Frammento 19 le quattro condizioni vengono usate per inizializzare la variabile `cond`: ogni volta che essa viene modificata, viene sommato a `value` un elemento della matrice. Se la condizione è falsa tale elemento della matrice risulta essere lo stesso che si vuole inizializzare durante l'esecuzione corrente dello *square step*: questo perché il valore da aggiungere all'indice per ottenere le coordinate dell'angolo deve essere moltiplicato alla condizione per evitare che vada a puntare a un indirizzo esterno al dominio; se è falsa, infatti, `cond` sarà 0, quindi il valore aggiunto verrà così annullato, poi è necessario moltiplicarla anche all'elemento per evitare che il valore di esso, ancora non inizializzato, vada a compromettere il risultato finale. Dopo questo passaggio la condizione deve essere sommata a `div` in modo da far sì che quest'ultima variabile sia incrementata solamente se `cond` è vera.

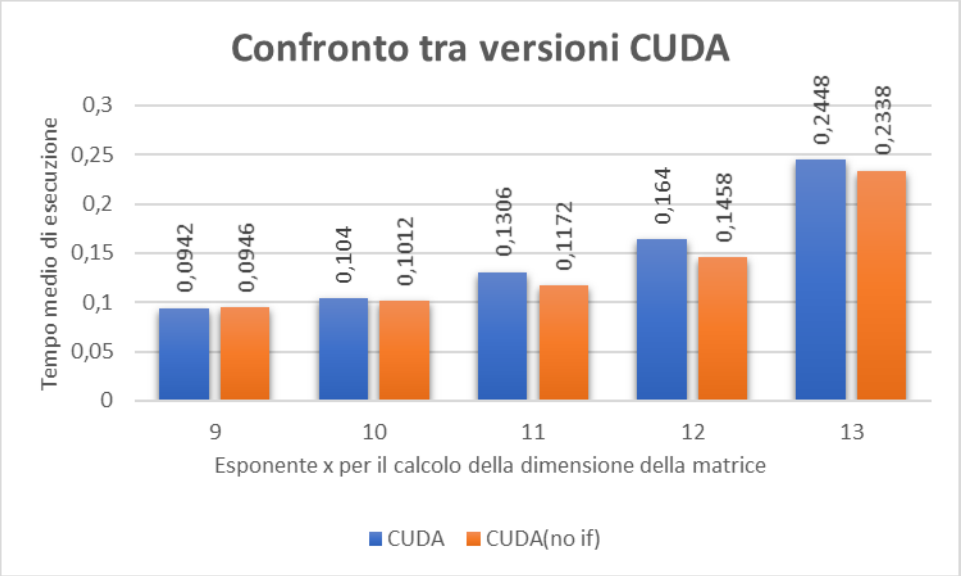

Grafico 4 - Grafico di confronto tra le due versioni CUDA descritte

Al termine di questa modifica sono stati raccolti i tempi per la nuova versione CUDA e sono stati messi a confronto con la versione precedente nel Grafico 4 sopra riportato. In esso gli istogrammi blu indicano i tempi medi ottenuti con l'esecuzione della versione iniziale

di CUDA, mentre quelli arancioni derivano dall'esecuzione della nuova versione senza blocchi condizionali. Si nota che la differenza tra le due versioni è minima, quindi anche con la modifica descritta la versione CUDA resta meno prestante della versione OpenMP finché $x < 13$.

Un altro miglioramento nelle prestazioni può essere ottenuto cambiando il modo di creare i valori pseudo-casuali: la libreria cuRAND, infatti, dà prestazioni migliori quando vengono creati blocchi di numeri casuali di grandi dimensioni, perché l'invocazione di poche creazioni anche di grandi quantità di valori è più efficiente rispetto a chiamate numerose per creare pochi numeri. Il generatore tramite algoritmo XORWOW, inoltre, necessita di una maggiore quantità di tempo la prima volta che viene invocato, ma questa può essere limitata utilizzando come metodo di ordinamento `CURAND_ORDERING_PSEUDO_SEEDED`: i metodi di ordinamento vengono utilizzati per specificare come i valori risultanti devono essere ordinati nella memoria globale della scheda grafica e questo specifico tipo di ordinamento è più leggero rispetto a `CURAND_ORDERING_PSEUDO_DEFAULT` usato solitamente, ma risulta essere anche più debole [28].

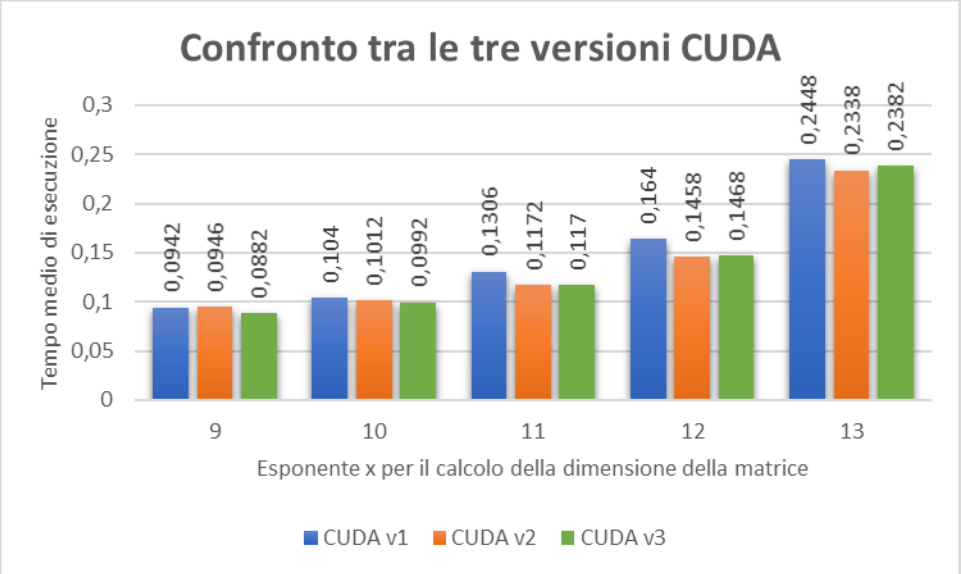

Grafico 5 - Grafico di confronto tra le tre versioni CUDA

Nel Grafico 5 vengono messe a confronto le prime due versioni di CUDA con una terza, in cui, invece di creare un generatore per ogni passo, ne viene creato solo uno all'inizio, utilizzato poi per calcolare

tutti i valori pseudo-casuali necessari contemporaneamente. Gli istogrammi blu e arancioni, come per il Grafico 4 a pagina 50, indicano i tempi per le prime due versioni, mentre quelli verdi rappresentano le medie dei tempi ottenuti con l'ultima versione descritta. Anche con quest'ultima modifica, tuttavia, non si notano miglioramenti molto significativi, soprattutto tra questa e la seconda implementazione, quindi le considerazioni fatte in precedenza restano valide anche per questa nuova versione.

Un'altra possibile causa delle scarse prestazioni ottenute con CUDA può essere data dal metodo di lettura della memoria globale utilizzato: ad ogni accesso, infatti, vengono inviati 32 o 128 byte consecutivi, quindi, per rendere più efficiente l'esecuzione di un programma, *threads* consecutivi dovrebbero accedere ad indirizzi di memoria consecutivi, in modo da dover chiedere meno accessi alla memoria globale perché è molto probabile che il contenuto dell'indirizzo seguente richiesto sia già stato inviato insieme a quello richiesto precedentemente. Per l'algoritmo in analisi, tuttavia, dovendo calcolare i valori centrali di una matrice utilizzando gli angoli di essa, è molto difficile che tali valori siano allocati in indirizzi consecutivi, quindi è l'algoritmo stesso a limitare l'utilizzo di CUDA, rendendo impossibile un miglioramento di questa condizione. Con i risultati ottenuti, tuttavia, si può dedurre che se fosse possibile aumentare ancora la dimensione del dominio, la versione CUDA risulterebbe comunque quella migliore da adottare perché è presumibile che i tempi ottenuti con OpenMP aumentino sensibilmente: osservando nuovamente il Grafico 3 a pagina 49, infatti, si nota che all'aumentare di x i tempi ottenuti serialmente o con OpenMP crescono di circa quattro volte, mentre quelli presi eseguendo la versione CUDA crescono molto più lentamente.

4. Morfologia matematica

4.1. Descrizione dell'algoritmo

La morfologia matematica [8] è una branca della matematica che si occupa dell'elaborazione delle immagini sfruttando la teoria degli insiemi. Con essa è possibile estrarre informazioni da un'immagine oppure filtrare quelle più importanti rimuovendo eventuali particolari di scarsa rilevanza. Solitamente la morfologia matematica lavora su immagini binarie, ovvero composte solamente da pixel bianchi e neri, tuttavia è possibile generalizzare il concetto per poter utilizzare anche immagini in scala di grigio. La morfologia matematica si basa sui concetti di *foreground* e *background*: il primo indica i pixel che fanno parte della forma rappresentata nell'immagine, mentre il secondo è tutto ciò che è nello sfondo. A seconda dell'immagine binaria su cui vogliamo eseguire operazioni di morfologia matematica, il *foreground* può essere composto da pixel di valore 0 (neri) oppure 255 (bianchi), mentre il *background* è l'inverso: l'insieme F dei pixel di *foreground* e F* di quelli di *background*, quindi, sono formulabili nel modo seguente:

Equazione 18 - definizione di foreground e background visti come insieme di pixel

$$F = \{p | p = [x, y]^T, img[x, y] = foreground\}$$
$$F^* = \{p | p = [x, y]^T, img[x, y] \neq foreground\}$$

dove img è l'immagine in esame, $foreground$ indica il colore scelto come colore principale dell'immagine (0 o 255), mentre $\neq foreground$ è il colore dello sfondo.

La maggior parte delle operazioni di morfologia matematica si basano sull'utilizzo di due operatori morfologici di base chiamati erosione e dilatazione; entrambi, per essere calcolati necessitano dell'utilizzo di un elemento strutturante.

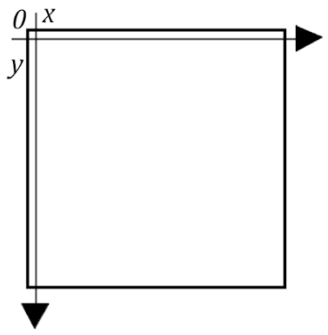

Figura 13 - Assi e origine di un'immagine

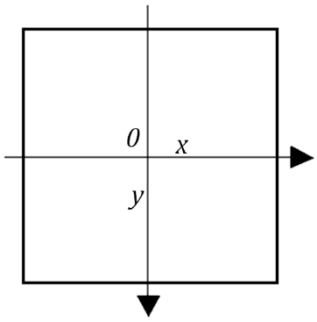

Figura 14 - Assi e origine di un elemento strutturante

Quest'ultimo è una piccola immagine binaria, solitamente quadrata e di lato dispari, considerata anch'essa come insieme di pixel di *foreground*; oltre a ciò, mentre di solito l'origine dell'immagine è considerata sul primo pixel in altro sinistra con l'asse y rivolto verso il basso, come mostrato in Figura 13, in Figura 14 si osserva che per un elemento strutturante l'origine deve essere al centro di esso, mentre gli assi restano orientati come per qualsiasi altra immagine. Nella seguente Equazione 19 viene riportata la formula dell'operazione di erosione.

Equazione 19 - formula dell'operazione di erosione

$$F \ominus S = \{q | (S)_q \subseteq F\}$$

In questa formula F è l'insieme dei pixel di *foreground* dell'immagine ed S è quello dell'elemento strutturante: l'erosione, quindi, restituisce come risultato l'insieme dei pixel dell'immagine per cui, se sovrapposti al centro dell'elemento strutturante, tutti i pixel di quest'ultimo sono contenuti in F; il *foreground* dell'immagine, quindi, viene ridotto ai soli pixel del risultato dell'erosione. L'operazione di dilatazione, invece, è definita nel modo seguente:

Equazione 20 - formula dell'operazione di dilatazione

$$F \oplus S = \{q | (S^r)_q \cap F \neq \emptyset\}$$

Dati F e S come descritti precedentemente, l'operazione di dilatazione dell'Equazione 20 restituisce l'insieme dei pixel dell'immagine per cui, se sovrapposti al centro della riflessione dell'elemento strutturante, almeno uno dei pixel di quest'ultima è sovrapposto all'immagine; il *foreground* dell'immagine, quindi, viene allargato perché alcuni pixel del *background*, se sovrapposti all'elemento strutturante riflesso, possono far parte del risultato della dilatazione. Da ciò si comprende che erosione e dilatazione sono una opposta all'altra perché in un caso l'immagine perde spessore, mentre nell'altro quest'ultimo viene ingrandito. Da questi due tipi di operazione morfologica è possibile derivarne altre, tra cui quelle di apertura e chiusura: la prima è usata per separare componenti debolmente connessi o eliminare piccoli

particolari, mentre la seconda serve per riempire buchi e rafforzare le connessioni tra componenti. L'operazione di apertura consiste nell'eseguire sull'immagine un'erosione seguita da una dilatazione, come mostrato nell'Equazione 21: il risultato, quindi, sarà un'immagine molto simile alla prima tranne che nei punti in cui la componente di *foreground* è più piccola dell'elemento strutturante utilizzato, perché a causa dell'erosione esse vengono completamente eliminate e la dilatazione non è in grado di ricostruire parti se di esse non resta neanche un pixel.

Equazione 21 - formula dell'operazione di apertura

$$F \circ S = (F \ominus S) \oplus S$$

Al contrario, la chiusura esegue prima la dilatazione, poi l'erosione e la sua formula è espressa nell'Equazione 22; con essa viene diminuita la presenza di buchi nel *foreground* perché grazie alla dilatazione questi vengono chiusi se di dimensione inferiore alla grandezza dell'elemento strutturante e una volta completamente chiusi l'erosione non può riaprirli.

Equazione 22 - formula dell'operazione di chiusura

$$F \cdot S = (F \oplus S) \ominus S$$

Le considerazioni appena fatte valgono per operazioni morfologiche eseguite su immagini binarie, ma esse sono anche la base per la morfologia matematica su immagini a scala di grigio [32]; anche per questo tipo di immagini, infatti, è possibile definire operazioni di dilatazione ed erosione, quindi anche di chiusura ed apertura, estendendo il modo in cui sono state definite per le immagini binarie per far sì che possano lavorare anche su più livelli di grigio. Per eseguire tali operazioni morfologiche su immagini in scala di grigio è necessario definire i concetti di elemento strutturante *flat* e *non-flat*: il primo è costituito da un'immagine binaria composta solo da pixel di *background* e *foreground*, mentre il secondo è, come l'immagine in entrata, un'immagine in scala di grigio, quindi con molti più valori possibili. Sia l'immagine in entrata che l'elemento strutturante, inoltre, a prescindere che esso sia *flat* o *non-flat*, per questo tipo di morfologia, invece di essere visti come insieme di pixel, vengono considerati rispettivamente come funzioni $f(x, y)$ e $b(x, y)$, dove $(x, y) \in \mathbb{Z}^2$ e indica le coordinate dei punti della matrice, mentre f e b sono funzioni

che associano ad ognuna di queste coppie di coordinate un valore preso dall'insieme dei livelli di grigio, che nel nostro caso è l'intervallo $[0,255] \in \mathbb{Z}$. Partendo da queste informazioni le operazioni di erosione e dilatazione possono essere ridefinite in due modi diversi a seconda del tipo di elemento strutturante utilizzato: se, ad esempio, quest'ultimo è *flat*, operazione di erosione viene definita come

Equazione 23 - formula dell'erosione in scala di grigio

$$[f \ominus b](x,y) = \min_{(s,t)\in b}\{f(x+s, y+t)\}$$

dove (s,t) sono le coordinate dei pixel di *foreground* dell'elemento strutturante, mentre (x,y) sono quelle dei pixel dell'immagine. Ad ogni coordinata (x,y), infatti, la erosione associa il valore minimo della porzione di immagine sovrapposta a b quando l'origine di quest'ultimo si trova in (x,y); per chiarire visivamente il discorso, in Figura 15 viene mostrata l'esecuzione di tale operazione. In questa figura l'immagine viene rappresentata come vista di lato mostrando sull'asse verticale i vari livelli di grigio possibili; ogni colonna mostrata indica in realtà un pixel la cui coordinata x è riportata sull'asse orizzontale e l'altezza di tale colonna rappresenta il colore di quel determinato pixel. La riga orizzontale verde è invece l'insieme dei pixel di *foreground* dell'elemento strutturante il cui centro è sovrapposto al pixel con coordinata x_0 dell'immagine. Detto ciò si può notare come l'elemento strutturante si "appoggi" al pixel con valore minore tra quelli intersecati da esso, appiattendo così anche gli altri all'altezza, quindi al colore, del minore. L'operazione di dilatazione, invece, viene definita come:

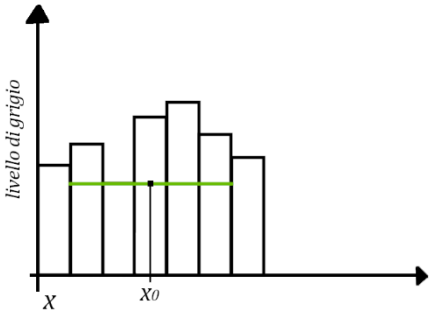

Figura 15 - Esecuzione dell'operazione di erosione in scala di grigio

Equazione 24 - formula dell'operazione di dilatazione in scala di grigio

$$[f \oplus b](x,y) = \max_{(s,t)\in b}\{f(x-s, y-t)\}$$

per cui, per ogni coordinata (x,y) dell'immagine, ad essa viene associato il valore massimo della porzione di immagine sovrapposta

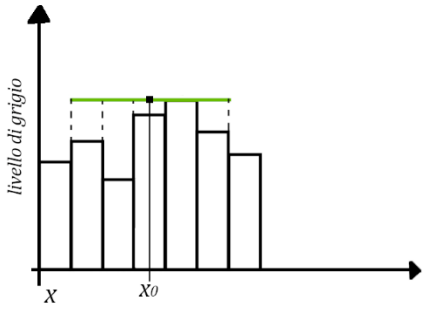

Figura 16 - Esecuzione dell'operazione di dilatazione in scala di grigio

alla riflessione dell'elemento strutturante il cui centro cade sul pixel di coordinata (x, y). Come per l'erosione, nella Figura 16 viene riportato un esempio di esecuzione dell'operazione di dilatazione: il grafico è analogo a quello di Figura 15, tuttavia, si nota che l'elemento strutturante, invece di intersecare i pixel dell'immagine, si "appoggia" al primo pixel che incontra, quindi quello più alto, portando poi gli altri al suo livello di grigio.

Quando, invece, si lavora con elementi strutturanti *non-flat*, non è possibile definire quali elementi fanno parte del *foreground* e quali sono solo di sfondo perché essi non sono più immagini binarie, bensì a scala di grigio; erosione e dilatazione, dunque, dovranno tenere conto anche del livello di grigio dell'elemento strutturante e le loro formule dovranno essere rispettivamente modificate nel modo seguente:

Equazione 25 - formula dell'operazione di erosione con elemento strutturante non-flat

$$[f \ominus b_{nf}](x,y) = \min_{(s,t)\in b_{nf}} \{f(x+s, y+t) + b_{nf}(s,t)\}$$

Equazione 26 - formula dell'operazione di dilatazione con elemento strutturante non-flat

$$[f \oplus b_{nf}](x,y) = \max_{(s,t)\in b_{nf}} \{f(x-s, y-t) + b_{nf}(s,t)\}$$

Da queste equazioni, tuttavia, si può osservare che l'utilizzo di elementi strutturanti *non-flat* può provocare risultati inconsistenti perché non è detto che il valore $f(x-s, y-t) + b_{nf}(s,t)$ sia un valore di f, quindi l'immagine risultato potrebbe essere composta da pixel con livelli di grigio non presenti nell'immagine in entrata.

Le definizioni di apertura e chiusura, infine, restano analoghe a quelle riportate rispettivamente in Equazione 21 ed Equazione 22 a pagina 55 con l'unica differenza che utilizzeranno le nuove formule di erosione e dilatazione di Equazione 23 ed Equazione 24 a pagina 56 in caso di elemento strutturante *flat*, o di Equazione 25 ed Equazione 26 altrimenti. In questa tesi verranno analizzate le esecuzioni di chiusura

ed apertura sui diversi canali di un'immagine a colori RGB, poiché essi possono essere visti come immagini in scala di grigio indipendenti l'una dall'altra; saranno inoltre messi a disposizione più elementi strutturanti *flat* con dimensioni diverse tra cui poter scegliere.

4.2. Sviluppo e implementazione

Per l'esecuzione delle operazioni di morfologia matematica è stata creata la classe astratta `MathematicalMorphology`, definita come nel seguente Frammento 20.

```
class HPCIMAGEPROCESSING_API MathematicalMorphology
{
public:
    MathematicalMorphology(FImage* image, int size);
    virtual ~MathematicalMorphology();
    virtual uint8* ExecuteOpeningOrClosing(bool
          isOpening) = 0;
protected:
    virtual void SplitChannels(uint8* redChannel,
          uint8* greenChannel, uint8* blueChannel,
          uint8 ghost) = 0;
    virtual void ExecuteErosion(uint8* input,
          uint8* output) = 0;
    virtual void ExecuteDilation(uint8* input,
          uint8* output) = 0;
    virtual void FillGhostCells(uint8* red, uint8*
          green, uint8* blue, uint8 value) = 0;
    FImage* input;
    StructuringElement structElem;
    Offset ErosionOffsets;
    Offset DilationOffsets;
private:
    void LoadStructuringElement(int size);
    void SetOffsets(Offset *offset, bool reflect);
    static const FString fileName;
    static const FString extension;
};
```

Frammento 20 - definizione della classe MathematicalMorphology

Dal frammento si nota la presenza del metodo astratto pubblico definito come `ExecuteOpeningOrClosing(bool isOpening)`: esso esegue l'operazione di apertura o di chiusura a seconda del valore del

parametro `isOpening` e nelle classi derivate da `MathematicalMorphology` sarà il metodo incaricato per l'invocazione degli altri. Oltre ai metodi virtuali implementati poi nelle classi derivate, questa classe contiene anche vari campi `protected` tra cui l'immagine di input passata dall'interfaccia grafica (`input`) un campo di tipo `StructuringElement` contenente le informazioni sull'elemento strutturante da utilizzare e due campi di tipo `Offset`. Il tipo di dato `StructuringElement` è una struttura definita all'interno *MathematicalMorphology.h* come mostrato nel Frammento 21.

```
struct StructuringElement
{
    uint8* element;
    int width;
    int height;
};
```

Frammento 21 - definizione della struttura StructuringElement

Essa contiene come campi due interi che indicano le dimensioni dell'elemento strutturante e un puntatore a interi a 8 bit contenente i byte che compongono quest'ultimo; per riempire i campi di questa struttura viene utilizzato il metodo `LoadStructuringElement(int size)` richiamato all'interno del costruttore della classe. Tale metodo cerca nel filesystem l'immagine da utilizzare come elemento strutturante di dimensione $size\ x\ size$, parametro passato al metodo dal costruttore che a sua volta lo ottiene dall'invocazione fatta nel metodo `ExecuteMMOperation(…)` della classe `TextureCreator`. Dovendo l'elemento strutturante essere impostato per ogni versione, il metodo `LoadStructuringElement(…)` viene lasciato privato perché non è necessario che sia definito per ogni classe derivata, ma basta richiamarlo nel costruttore della classe base e far sì che le derivate chiamino quest'ultimo nel loro. All'interno del costruttore della classe vengono inizializzati anche i campi di tipo `Offset` tramite il metodo privato `SetOffsets(Offset* offset, bool reflect)`. Anche quest'ultimo tipo di dato è una struttura, riportata nel seguente Frammento 22.

```
struct Offset
{
    int* offsets;
    int count = 0;
};
```

Frammento 22 - definizione della struttura Offset

`Offset` è composta solo da due campi: uno è un puntatore ad interi contenente i vari *offsets* utili per eseguire le operazioni di erosione e dilatazione, l'altro è di tipo intero e conterrà il numero effettivo di elementi aggiunti al puntatore precedente tramite il metodo `SetOffsets(…)`. Quest'ultimo prende come parametri la variabile di tipo `Offset` da inizializzare e un booleano `reflect`, con cui viene specificato se l'elemento strutturante deve essere riflesso, in caso di dilatazione, oppure no, se quella variabile di tipo `Offset` deve essere usata per eseguire l'erosione. Il campo `offsets` dell'ultima struttura descritta, infatti, deve contenere la distanza che intercorre tra un pixel dell'elemento strutturante di *foreground* dal centro di quest'ultimo, facendo in modo che queste distanze facciano riferimento alle dimensioni dell'immagine di input per poterle utilizzare per trovare i pixel dell'intorno di ognuno di quelli che costituiscono l'immagine. Ciò è necessario per il modo in cui una matrice viene allocata in memoria: come esempio, in Figura 17 viene riportata l'applicazione di un elemento strutturante $3x3$ sull'elemento di una matrice $9x8$.

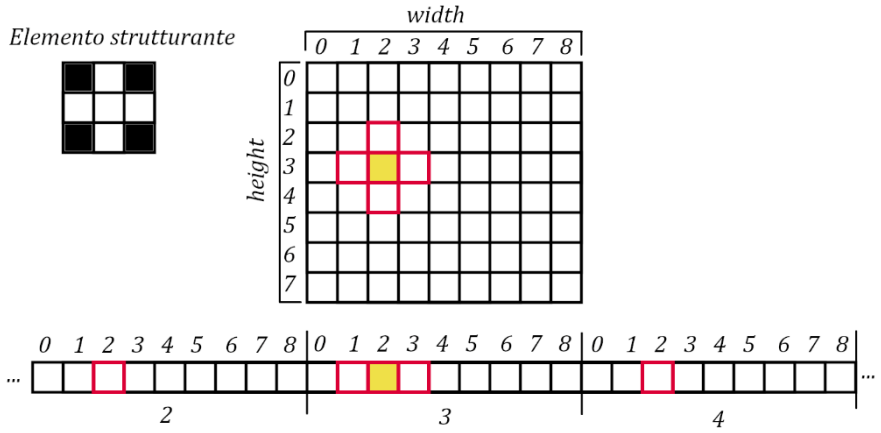

Figura 17 - Esempio di applicazione di un elemento strutturante a un'immagine

Nell'elemento strutturante riportato, i pixel di *foreground* sono quelli di colore bianco, perciò, dato l'elemento giallo della matrice come punto

sovrapposto all'origine dell'elemento strutturante, i pixel da utilizzare per calcolare il valore di tale cella sono quelli dell'intorno evidenziato in rosso. Come mostrato sotto, tuttavia, le celle di una matrice vengono allocate in memoria per righe consecutive, quindi se i pixel dell'intorno ricadono su righe diverse la distanza che intercorre tra quei pixel e quello che deve essere modificato dipende dalla larghezza della matrice, in questo caso pari a 9. Dall'esempio si può notare che i punti di *foreground* dell'elemento strutturante sono $(0,0)$, $(0,-1)$, $(0,1)$, $(-1,0)$ e $(1,0)$, considerando l'origine al centro di esso, tuttavia per scorrere all'interno del codice tali punti conviene considerare l'elemento strutturante come una qualsiasi altra matrice con l'origine nel primo elemento della prima riga; da ciò i punti appena citati devono essere considerati come segue:

Equazione 27 - coordinate dei punti di foreground dopo la traslazione dell'origine dell'elemento strutturante

$$(0,0) \rightarrow (1,1)$$
$$(0,-1) \rightarrow (1,0)$$
$$(0,1) \rightarrow (1,2)$$
$$(-1,0) \rightarrow (0,1)$$
$$(1,0) \rightarrow (2,1)$$

Con queste nuove coordinate, per eseguire l'operazione di erosione il campo `ErosionOffset` verrà riempito con $count = 5$, dato dal numero di punti di *foreground* dell'elemento strutturante, e con le distanze calcolate come:

Equazione 28 - formula per calcolare le distanze dal centro dell'elemento strutturante per operazioni di erosione

$$\left(y - \frac{h}{2}\right) * width + x - \frac{w}{2}$$

dove x indica la coordinata orizzontale con origine traslato dell'elemento strutturante, y quella verticale, w è la larghezza dell'elemento strutturante, h la sua altezza e $width$ la larghezza dell'immagine su cui eseguire l'erosione; $x - w/2$ e $y - h/2$ sono invece le traslazioni usate per tornare a riferirsi alle coordinate corrette con origine posto al centro. Le distanze così calcolate possono poi essere utilizzate per qualsiasi immagine di larghezza $width$, perciò, dato che il programma effettua operazioni di apertura o chiusura in scala

di grigio sui diversi canali di un'immagine RGB, `ErosionOffset` può essere inizializzato una sola volta e poi essere utilizzato su tutti i canali colore perché essi avranno tutti la stessa dimensione.

Per l'operazione di dilatazione, invece, dato che l'elemento strutturante deve essere riflesso rispetto al suo centro, le distanze dovranno essere calcolate come segue:

Equazione 29 - formula per calcolare le distanze dal centro dell'elemento strutturante per operazioni di dilatazione

$$\left(y_{max} - y - \frac{h}{2}\right) * width + x_{max} - x - \frac{w}{2}$$

Nell'Equazione 29 vengono utilizzati anche i simboli x_{max} e y_{max} che indicano l'ultima coordinata dell'elemento strutturante rispettivamente sugli assi x e y, in modo che esso risulti come ribaltato su entrambi gli assi: per il punto $(-1,0)$ la distanza dal centro risulterà essere come quella del punto $(1,0)$ in assenza di riflessione, viceversa per il punto $(1,0)$, mentre per i punti $(0,-1)$ e $(0,1)$ essa sarà equivalente rispettivamente a quelle dei punti $(0,1)$ e $(0,-1)$. Essendo l'elemento strutturante di esempio simmetrico, le distanze calcolate per le operazioni di erosione e per quelle di dilatazione sono le stesse poiché anche riflettendolo i punti che prima erano considerati di *foreground* restano tali; tuttavia, se non ci fosse questa simmetria, è possibile che per la dilatazione vengano usate distanze diverse perché un punto di *foreground*, una volta riflesso, non è detto che vada a combaciare con la posizione di un altro punto di *foreground* presente prima della riflessione. In seguito, verrà spiegato come vengono utilizzate le distanze appena calcolate per trovare i pixel dell'intorno di una cella della matrice.

4.2.1. Versione seriale

L'implementazione seriale di questo algoritmo è contenuta nella classe `SerialMMorphology`, derivata da `MathematicalMorphology`; in essa vengono implementati i metodi astratti presenti nella classe base e vengono eseguite le varie operazioni richieste. Nel seguente pseudo-codice del Frammento 23 viene riportato ciò che deve essere svolto dal metodo `ExecuteOpeningOrClosing(bool isOpening)`.

```
funzione ExecuteOpeningOrClosing(bool isOpening):
    intero a 8 bit
    dividi canali
    se isOpening = vero allora
          esegui erosione canale rosso
          esegui dilatazione canale rosso
          esegui erosione canale verde
          esegui dilatazione canale verde
          esegui erosione canale blu
          esegui dilatazione canale blu
    altrimenti
          esegui dilatazione canale rosso
          esegui erosione canale rosso
          esegui dilatazione canale verde
          esegui erosione canale verde
          esegui dilatazione canale blu
          esegui erosione canale blu
    fine se
fine funzione
```

Frammento 23 - pseudo-codice del metodo ExecuteOpeningOrClosing(...)

La prima cosa che deve essere fatta è dividere i canali dell'immagine invocando il metodo `SplitChannels(uint8* redChannel, uint8* greenChannel, uint8* blueChannel, uint8 ghost)`, poi i canali così creati possono essere usati come input per l'esecuzione di chiusura o apertura. Dato che queste operazioni vengono eseguite alternando dilatazione ed erosione e che esse calcolano i valori dei diversi pixel controllando i valori di quelli attorno, per gli elementi più esterni dell'immagine è possibile che i pixel dell'intorno cadano fuori dal dominio. Riferendosi all'esempio di Figura 17 a pagina 60, infatti, l'intorno di un pixel è dato da quelli posti sulla stessa riga ma sulla colonna precedente e successiva e quelli sulla stessa colonna ma sulla riga prima e quella dopo: per la prima e ultima riga e per la prima e ultima colonna, quindi, parte dell'intorno cadrebbe all'esterno della matrice. Per risolvere questo problema è stato deciso di allocare per ogni canale, oltre allo spazio effettivo dell'immagine, anche una *ghost area* che aggiunga righe e colonne prima e dopo il dominio della matrice. Questa *ghost area* deve aggiungere prima della prima riga e dopo l'ultima un numero di righe pari alla metà dell'altezza dell'elemento strutturante, perché sovrapponendo il centro di quest'ultimo a un pixel della prima (o ultima) riga, $h/2$ è il numero di

righe al di sopra (o al di sotto) di tale centro. Infine, le colonne di *ghost cells* precedenti alla prima colonna e successive all'ultima, per lo stesso motivo, devono essere metà della larghezza dell'elemento strutturante. Avendo aggiunto righe e colonne in più ai diversi canali, il metodo `SplitChannels(…)` non può semplicemente scorrere gli elementi dell'immagine dal primo all'ultimo perché altrimenti i canali verrebbero inizializzati partendo dalle celle della *ghost area*: vengono quindi inizializzate nuove variabili locali al metodo con cui definire i valori iniziali e finali di righe e colonne che dovranno contenere gli effettivi pixel dei canali.

```
int firstRow = (this->structElem.height - 1) / 2;
int firstCol = (this->structElem.width - 1) / 2;
int lastRow = this->input->SizeY + firstRow;
int lastCol = this->input->SizeX + firstCol;
int width = this->input->SizeX
    + this->structElem.width-1;
int height = this->input->SizeY
    + this->structElem.height-1;
```

Frammento 24 - frammento di codice di SplitChannels(...)

Nel Frammento 24 sono riportate le variabili citate in precedenza: `firstRow` indica la prima riga di pixel del canale, quelle prima di essa sono righe della *ghost area*; in maniera equivalente è definita `firstCol`, rappresentante la prima colonna utilizzabile per l'inizializzazione della matrice. Come già detto tali variabili vengono calcolate rispettivamente come la metà dell'altezza e la metà della larghezza dell'elemento strutturante, mentre `lastRow` e `lastCol` vengono calcolate aggiungendo alle dimensioni effettive dell'immagine i valori ottenuti per `firstRow` e `firstCol`: essendo inizializzate in questo modo queste due ultime variabili saranno in realtà gli indici della prima riga e colonna delle *ghost cells* finali. Infine, le variabili `width` ed `height` indicano la larghezza e l'altezza della matrice compresa la *ghost area*. I for con cui vengono iterati i pixel, quindi, partiranno dall'indice 0 sia per le righe che per le colonne e arrivare fino all'ultimo, ovvero `height-1` per le righe e `width-1` per le colonne; in questo modo, ogni elemento dei canali da inizializzare sarà all'indice $i * width + j$, con i come variabile di iterazione delle righe e j per le colonne. All'interno del ciclo verrà poi eseguito un controllo su tali

coordinate perché per $i < firstRow,\ i \geq lastRow,\ j < firstCol,$ oppure $j \geq lastCol,$ i pixel corrispondenti fanno parte della *ghost area*, quindi devono essere inizializzati con il valore del parametro `ghost` passato al metodo; in tutti gli altri casi invece i pixel corrisponderanno a un elemento dell'immagine. Questi ultimi devono essere iterati partendo dal primo, perché nell'immagine di input non è presente la *ghost area*, quindi le sue dimensioni reali corrispondono a quelle effettive del dominio: per poter quindi associare ai vari elementi dei canali i pixel dell'input, questi ultimi devono fare riferimento all'indice $(i - firstRow) * sizeX + j - firstCol,$ che corrisponde a scorrere l'immagine di lato $sizeX$ dal primo pixel all'ultimo. La necessità del parametro ghost nel metodo `SplitChannels(…)` nasce dal fatto che tale valore non può essere definito a priori perché esso dipende dall'operazione morfologica che deve essere eseguita su quello specifico canale. Se vogliamo eseguire un'operazione di apertura, infatti, i canali vengono utilizzati come input dell'operazione di erosione, quindi, dovendo in essa calcolare il minimo dell'intorno, tutte le *ghost cells* di ognuno di essi vengono inizializzate a 255, ovvero il valore massimo possibile che quindi sarà sempre maggiore o uguale ai valori dei pixel interni alla matrice. Il risultato di tale operazione deve essere salvato su canali diversi rispetto a quelli di input, quindi nasce la necessità di creare tre ulteriori matrici la cui inizializzazione viene eseguita dall'operazione di erosione. Se l'obiettivo finale fosse solo l'erosione, questi nuovi canali potrebbero avere le dimensioni reali dell'immagine, tuttavia, dovendo eseguire un'apertura, essi saranno l'input della successiva operazione di dilatazione, quindi dovranno contenere anche loro la stessa quantità di *ghost cells* dei primi: esse verranno inizializzate tramite la funzione `FillGhostCells (uint8* red, uint8* green, uint8* blue, uint8 value)` in cui, per l'esempio corrente, `value` sarà 0, poiché la dilatazione calcola il valore del pixel utilizzando il valore massimo dell'intorno, quindi per evitare che i valori delle *ghost cells* influenzino il risultato, esse vengono inizializzate con il minore valore possibile. Al contrario, per l'operazione di chiusura, dato che prima vengono eseguite le dilatazioni e poi le erosioni, le *ghost cells* dei canali di input dovranno essere inizializzate a 0, mentre quelle degli altri a 255; questo fa sì che lo pseudo-codice del Frammento 23 non sia del tutto corretto: dovendo

inizializzare la *ghost area* dei canali in maniera diversa in base all'operazione da svolgere, la divisione di essi e l'inizializzazione delle *ghost cells* degli altri dovrà essere fatta all'interno dei blocchi condizionali, per poter così specificare i giusti valori da assegnare a questi elementi.

Proseguendo con l'analisi della versione seriale si nota che le implementazioni dei metodi `ExecuteErosion(…)` ed `Execute-Dilation(…)` sono molto simili tra loro: entrambi i metodi prendono come parametri un puntatore ad interi ad 8 bit che corrisponde al canale su cui eseguire l'operazione e un altro che invece conterrà il risultato. Come per il metodo `SplitChannels(…)`, anche in questi sono presenti variabili locali analoghe a quelle riportate nel Frammento 24, in questo caso, tuttavia, i cicli che iterano le righe e le colonne partono proprio dal valore di `firstRow` e `firstCol` fino ad arrivare agli indici precedenti a `lastRow` e `lastCol`, escludendo completamente le coordinate della *ghost area* dai calcoli. All'interno di questi cicli, poi, per ogni pixel viene eseguito un ulteriore ciclo for come segue:

```
for (int i = 0; i < this->ErosionOffsets.count;
    i++)
{
    if (in[row*width + col + this->ErosionOffsets.
          offsets[i]] < minValue)
    {
          minValue = in[row*width + col
                + this->ErosionOffsets.offsets[i]];
    }
}
```

Frammento 25 - porzione di codice estratta dal metodo ExecuteErosion(...)

Il Frammento 25 mostra una porzione di codice estrapolata dal metodo `ExecuteErosion(…)` e riporta il ciclo che, per ogni pixel dell'immagine, calcola il valore minimo scorrendo gli elementi del campo `offsets` della struttura `ErosionOffsets` precedentemente inizializzato come descritto in **4.2 Sviluppo e implementazione** da pagina 58; da quest'ultimo frammento si nota anche che per trovare i pixel dell'intorno, ogni valore di `offsets` viene sommato all'indice del pixel corrente. Trovato poi il minore tra i valori dell'intorno, esso viene assegnato all'elemento della matrice risultato corrispondente agli

indici dell'iterazione in esecuzione. Nel metodo `ExecuteDilation(…)`, infine, viene eseguito un ciclo molto simile a quello del Frammento 25 che, invece di utilizzare le distanze contenute nella struttura `ErosionOffset`, usa quelle di `DilationOffset` e, invece di cercare il minimo, cerca il massimo e aggiorna con esso l'elemento della matrice risultato.

Una volta che queste operazioni sono state eseguite per ogni canale, questi devono essere ricombinati insieme tramite il metodo `ComposeImage(…)`: anch'esso fa riferimento a variabili uguali a quelle del Frammento 24 a pagina 64 per far sì che l'immagine finale non contenga la *ghost area*. Per poter ricostruire l'immagine a colori ogni pixel di un canale deve essere alternato a quelli corrispondenti all'interno degli altri in base al formato immagine da utilizzare: il formato scelto in questa tesi è BGRA8, quindi ogni pixel dell'immagine è formato da un byte recuperato dal canale blu, uno dal canale verde, uno dal rosso e uno dal canale *alpha* per l'eventuale trasparenza. Nel frammento seguente viene riportato il contenuto del ciclo con cui viene ricomposta l'immagine.

```
//blue channel
output[j] = blueChannel[i*width + k];
j++;
//green channel
output[j] = greenChannel[i*width + k];
j++;
//red channel
output[j] = redChannel[i*width + k];
j++;
//alpha channel
output[j] = ALPHA;
j++;
```

Frammento 26 - porzione di codice con cui viene ricomposta l'immagine a colori

In tale frammento, le variabili `i` e `k` vengono utilizzate nei for per iterare gli elementi dei vari canali, mentre `j` è un ulteriore indice, precedentemente definito e inizializzato a 0, che indica i vari elementi del puntatore `output` che conterrà i byte che compongono l'immagine a colori. Per ogni iterazione verranno quindi inizializzati quattro byte della nuova immagine, i primi tre con i pixel dei tre canali in posizione

(i, k), mentre l'ultimo, quello riferito al canale *alpha*, sempre con valore 255 per far sì che l'immagine sia totalmente senza trasparenza.

4.2.2. Versione parallela con OpenMP

Partendo dalla versione seriale appena descritta, nella classe `OpenMPMMorphology` viene creata una prima versione parallela usando OpenMP. Anche in questa classe il metodo principale è `ExecuteOpeningOrClosing(…)`, perciò esso può essere utilizzato come punto di partenza per lo studio sulla parallelizzazione, osservando lo pseudo-codice riportato nel Frammento 23 a pagina 63. Da esso si nota che la prima istruzione da eseguire è l'invocazione del metodo `SplitChannels(…)` con cui dividere i canali dell'immagine: questa operazione può essere svolta in maniera parallela utilizzando `#pragma omp parallel for` senza problemi perché la scrittura di un elemento di uno dei canali non dipende né da altri elementi dello stesso né da quelli degli altri, poiché è sufficiente leggere il valore corrispondente del puntatore a `FColor` ricavato dall'immagine di input. Utilizzando le classi messe a disposizione da Unreal Engine, infatti, data un'immagine di tipo `FImage` [33], è possibile trasformarla in un puntatore a `FColor` [34] che indicherà quindi un elenco di colori composti da canali di 8 bit di dimensione; per ogni elemento di tale elenco, dunque, è possibile ricavare tramite i campi `R`, `G` e `B` di `FColor`, i vari canali di cui si ha necessità. Dato che l'immagine può essere di formati diversi, essa è composta da pixel la cui dimensione dipende da quanti byte si vogliono usare per ogni canale e da come è stato scelto di ordinare questi ultimi per ogni pixel: per poter quindi trasformare in maniera corretta gli elementi dell'immagine in `FColor` è necessario utilizzare la funzione di `FImage` specifica per il formato scelto, altrimenti i vari canali potrebbero essere interpretati male. In questo caso, quindi, la funzione da utilizzare è `AsBGRA8()`, poiché il formato usato è BGRA con 8 bit per ogni canale, come accennato precedentemente in **3.2.1 Versione seriale**. La divisione dei canali, tuttavia, può essere eseguita parallelamente all'inizializzazione della *ghost area* degli altri tre canali di output perché le sei matrici sono completamente indipendenti tra loro: l'esecuzione di `SplitChannels(…)` e di `FillGhostCells(…)`, quindi, può essere svolta tramite sezioni. In OpenMP, infatti, è possibile definire tramite `#pragma omp sections` porzioni di codice

che possono essere eseguite contemporaneamente in varie sezioni, ognuna definita all'interno di un blocco specificato come `#pragma omp section` [7]. Ogni sezione viene eseguita da un singolo *thread* in parallelo con le altre, quindi, se si sceglie di eseguire `SplitChannels(…)` all'interno di una di esse non è possibile parallelizzare i cicli come detto precedentemente. Un blocco definito con la direttiva `#pragma omp sections` deve essere contenuto all'interno di una regione parallela creata tramite `#pragma omp parallel` e non necessita di barriere esplicite alla fine dell'elenco di sezioni perché al termine di tale blocco è già presente una barriera implicita su cui, una volta terminata l'esecuzione della propria sezione, ogni *thread* aspetta che anche tutti gli altri abbiano finito. Se, invece di utilizzare il parallelismo offerto dalle sezioni, si preferisce mantenere il parallelismo dei cicli di `SplitChannels(…)`, è possibile parallelizzare allo stesso modo anche le iterazioni di `FillGhost-Cells(…)` perché anch'esse sono una indipendente dall'altra; per evitare di creare più *pool* di *threads*, tuttavia, come per l'implementazione OpenMP dell'algoritmo Diamond-Square, è preferibile richiamare una sola volta `#pragma omp parallel` all'inizio del metodo `ExecuteOpeningOrClosing(…)` e successivamente parallelizzare i vari cicli dei due metodi citati in precedenza con la direttiva `#pragma omp for`.

Una volta inizializzati i vari canali devono essere eseguite le operazioni di erosione e dilatazione: esse, riferendosi una al risultato dell'altra, devono essere necessariamente eseguite in sequenza altrimenti si rischierebbe di calcolare il valore di un pixel considerando un intorno non ancora completamente inizializzato; dovendo però essere eseguite per tutti e tre i canali dell'immagine ed essendo questi indipendenti l'uno dall'altro, è possibile racchiudere le operazioni di erosione e di dilatazione eseguite sullo stesso canale all'interno di una sezione, così da crearne una per ogni canale per cui eseguire l'apertura o la chiusura. Anche i cicli di `ExecuteErosion(…)` ed `ExecuteDilation(…)`, tuttavia, possono essere parallelizzati, dunque anche in questo caso si può scegliere se utilizzare le sezioni oppure sfruttare il parallelismo dei cicli; il metodo `ComposeImage(…)`, invece, può essere parallelizzato solamente suddividendo le varie iterazioni dei cicli tra i diversi *threads*, utilizzando quindi `#pragma omp for`, poiché esso può essere eseguito

solamente al termine di tutte le altre operazioni, quindi non può essere contenuto in nessun blocco di sezioni. La definizione di quest'ultimo metodo, inoltre, presenta differenze rispetto a quella del metodo della versione seriale: mentre prima esso restituiva l'immagine finale, ora si è scelto di allocare tale puntatore prima dell'invocazione al metodo e di passarglielo poi come parametro: questo evita che tale puntatore venga allocato più volte da ogni *thread* della regione parallela come variabile privata, mentre in questo modo risulta condiviso tra tutti i *threads* del *pool*. Nella versione seriale, inoltre, i pixel dell'immagine risultato venivano iterati con un indice `j` inizializzato a 0 prima del ciclo e incrementato per ogni byte di output, come mostra il Frammento 26 a pagina 67; se venisse utilizzato anche qui lo stesso metodo, tuttavia, `j` dovrebbe essere considerata variabile condivisa tra i vari *threads*, perché se fosse privata ognuno di essi andrebbe a scrivere sempre sugli stessi indirizzi di memoria dell'immagine, poiché per tutti partirebbe da 0. I *threads* che eseguono i cicli di questo metodo però non vengono creati all'interno di esso ma prima, all'invocazione di `#pragma omp parallel`, quindi in realtà il metodo viene invocato da tutti i *threads*, perciò ogni sua variabile locale è sempre considerata privata, poiché non viene definita prima dell'inizio della regione parallela ma già al suo interno. Per risolvere questo problema è stato deciso di cambiare il tipo di indice con cui vengono scorsi i vari byte che compongo l'immagine: questo nuovo indice è calcolato come

Equazione 30 - indice di scorrimento dei valori dell'immagine di output

$$index * CHANNELS + j$$
$$con\ index = (i - firstRow) * sizeX + k - firstCol$$

La variabile `index`, infatti, viene inizializzata considerando la dimensione dell'immagine esclusa la *ghost area* presente nei canali, tuttavia, mentre in questi ultimi ogni byte corrisponde a un pixel, nell'immagine di output ognuno di essi è composto da quattro byte, uno per ogni colore, quindi la sua dimensione effettiva è quattro volte quella del semplice canale. L'indice `index*CHANNELS`, inoltre, indicherà per ogni iterazione il primo byte che compone un pixel dell'immagine, quindi quello del primo canale definito dal formato di immagine scelto: in questo caso tale canale è il blu, mentre per ottenere i pixel degli altri canali è necessario accedere agli indirizzi di memoria successivi a quelli

indicati da tale indice; per questo motivo la variabile `j` può essere inizializzata a 0 per ogni *thread* senza problemi, poiché per tale valore potrà inizializzare per ogni pixel il byte riferito al colore blu, mentre con i successivi incrementi potrà inizializzare anche quelli degli altri canali. Nel codice finale di questa versione parallela è stato deciso di evitare l'utilizzo delle sezioni e di utilizzare solamente il parallelismo delle iterazioni dei cicli, in questo modo `SplitChannels(…)` e `FillGhostCells(…)` verranno eseguiti uno di seguito all'altro, proprio come le varie operazioni svolte sui diversi canali. Tale scelta verrà spiegata nel dettaglio nella successiva sezione **4.3 Analisi delle prestazioni**.

4.2.3. Versione parallela con CUDA

Per quanto riguarda la versione parallela implementata con CUDA, essa è contenuta nella classe `CudaMMorphology` al cui interno viene utilizzata la libreria *MathematicalMorphologyCuda*. Essa contiene la classe `CudaMathMorphology` al cui interno è definito il metodo statico `ExecuteOpeningOrClosing(int structWidth, int structHeight, uint8_t* image, int width, int height, int* erOffset, int erCount, int* dilOffset, int dilCount, bool isOpening)` invocato nell'omonimo metodo di `CudaMMorphology` per l'esecuzione dell'operazione morfologica scelta. I parametri `structWidth` e `structHeight` indicano rispettivamente la larghezza e l'altezza dell'elemento strutturante scelto, perché, invece di passare a tale metodo l'intero elemento strutturante, questo viene processato dalle funzioni di `CudaMMorphology`, passando così al metodo in questione solamente le informazioni che gli sono utili, come le dimensioni dell'elemento e gli *offsets* per l'esecuzione delle varie operazioni. Questi ultimi vengono passati tramite i parametri `erOffset` e `dilOffset`, contenenti le distanze calcolate per eseguire rispettivamente l'erosione e la dilatazione, mentre il numero di elementi che contengono è indicato da `erCount` e `dilCount`. Il parametro `image`, invece, contiene i vari pixel dell'immagine, mentre `width` ed `height` sono la sua larghezza e la sua altezza; infine viene passato anche il booleano `isOpening` per poter così decidere all'interno del metodo quale operazione eseguire. Mentre nelle altre versioni si lavorava con l'immagine di tipo `FImage`, qui essa

viene passata direttamente come puntatore a una matrice di pixel: questo è dovuto al fatto che `CudaMathMorphology` è una classe definita in una libreria esterna ad Unreal Engine, quindi in essa non possono essere inclusi i file di intestazione delle classi interne al motore grafico. Né `FImage` né `FColor` possono dunque essere utilizzati nei metodi di questa classe, quindi, per passare i pixel dell'immagine al suo metodo, il parametro `image` dovrà essere inizializzato durante l'invocazione come `input->RawData.GetData()`, dove `input` è l'immagine. `RawData`, infatti, è un campo della classe `FImage` di tipo `TArray64<uint8>`, ovvero un vettore dinamico di interi ad 8 bit; la classe generica `TArray<>` [35] viene utilizzata per creare vettori dinamici di oggetti del tipo specificato tra parentesi angolari; essa mette a disposizione il metodo `GetData()` che restituisce il puntatore agli elementi contenuti, quindi in questo caso agli interi a 8 bit che compongono i pixel dell'immagine. All'interno del metodo tali pixel verranno poi suddivisi in canali invocando il kernel `SplitChannels(uint8_t* image, uint8_t* red, uint8_t* green, uint8_t* blue, int width, int height, int structWidth, int structHeight, int ghost)`: il parametro `image` indica l'immagine di input, `red`, `green` e `blue` i diversi canali su cui effettuare le operazioni morfologiche, `ghost` è il valore con cui inizializzare la *ghost area*, mentre le altre variabili sono analoghe a quelle passate come parametri a `ExecuteOpeningOrClosing(…)`. Tutti i kernel che verranno lanciati per eseguire le operazioni richieste vengono eseguiti da un numero di *threads* pari al numero dei byte che compongono i diversi canali, comprese le *ghost cells*: idealmente, quindi, i *threads* dovrebbero far parte di un blocco di dimensione $sizeXxSizeY$, dove:

Equazione 31 - formule con cui vengono calcolate le dimensioni totali dell'immagine compresa di ghost cells

$$sizeX = width + structWidth - 1$$
$$sizeY = height + structHeight - 1$$

Come già detto, tuttavia, il numero di *threads* in un blocco è limitato e come per l'altro algoritmo anche qui viene impostato a 512: la dimensione massima dei blocchi è quindi $22x22 = 484$, poiché con la grandezza successiva $23x23 = 529$ il blocco sarebbe troppo grande.

Per poter realizzare il numero totale di *threads* richiesti i blocchi saranno poi disposti in una griglia con righe lunghe $(sizeX + 22 - 1)/22$ e con colonne di $(sizeY + 22 - 1)/22$ elementi: tali formule derivano dal fatto che non è detto che la lunghezza di una riga o di una colonna siano divisibili per 22, quindi effettuando la divisione semplice si rischierebbe di perdere dei *threads*. Come per le versioni precedenti, il kernel `SplitChannels(…)` inizializza sia i valori dei pixel dei canali che fanno parte della matrice, sia quelli della *ghost area*, controllando l'indice del *thread* incaricato di inizializzare quel determinato elemento: per evitare la programmazione divergente data dalla presenza di costrutti condizionali, quindi, vengono definite all'interno del kernel le variabili `img` ed `halo`, la prima contenente la condizione per cui l'elemento deve essere impostato con un byte dell'immagine, l'altra per la condizione per cui il pixel è da considerarsi una cella "fantasma". All'interno del kernel, dunque, i vari elementi dei canali verranno inizializzati come mostrato nel Frammento 27.

```
…
int index = y * sizeX + x;
int i = (y - firstY)*width + x - firstX;
i = i * img + 0 * halo;
…
blue[index] = image[i*CHANNELS + j] * img + ghost *
    halo;
j++;
green[index] = image[i*CHANNELS + j] * img + ghost
    * halo;
j++;
red[index] = image[i*CHANNELS + j] * img + ghost *
    halo;
…
```

Frammento 27 - porzione di codice estrapolato dal kernel SplitChannels(...)

I pixel dei canali colore hanno come indice `index`, dato dalle coordinate del *thread* corrente e dalla dimensione totale della riga calcolata come nell'Equazione 31; se il *thread* deve inizializzare uno degli elementi effettivi del canale la variabile `img` sarà vera, mentre `halo` sarà falsa, così il valore del pixel sarà dato solamente dal valore del byte di `image` all'indice `i*CHANNELS+j`, altrimenti il pixel verrà inizializzato con `ghost`, scelto in base all'operazione da svolgere.

L'indice di scorrimento di `image`, invece, è dato da `i`, calcolato come riportato nel frammento, moltiplicandolo al numero di canali che compongono l'immagine, ovvero quattro, a cui viene aggiunto poi il valore di j, come per lo scorrimento della matrice risultato eseguito da `ComposeImage(…)` per la versione OpenMP. Dal Frammento 27, inoltre, si nota che dopo l'inizializzazione della variabile `i`, essa viene subito modificata come se stessa moltiplicata per `img` sommata poi a 0 moltiplicato per `halo`. Questa nuova inizializzazione è dovuta al fatto che, non utilizzando costrutti condizionali, il *thread* tenta di accedere a `image` anche quando l'indice con cui viene scorsa non fa parte del suo dominio, causando quindi eccezioni; impostando `i` usando le variabili `img` ed `halo`, invece, quando le coordinate dei *threads* fanno riferimento a una *ghost cell* l'indice di scorrimento di `image` risulta essere sempre 0, che ovviamente è sempre un indice corretto all'interno della matrice. Il valore 0 può essere scambiato con qualsiasi altro valore purché all'interno del dominio dell'immagine perché in ogni caso l'elemento in quella posizione non verrà utilizzato realmente per il calcolo del pixel del canale, dato che la condizione `img` risulta falsa.

Utilizzando lo stesso numero di *threads* viene poi lanciato il kernel `SetDefault(uint8_t* red, uint8_t* green, uint8_t* blue, int width, int height, uint8_t value)`, dove `red`, `green` e `blue` sono i canali che conterranno il risultato delle operazioni eseguite su quelli derivati dall'immagine, `value` è il valore da dare alle *ghost cells*, mentre `width` ed `height` sono le dimensioni dell'immagine, sebbene stavolta vengono direttamente considerate quelle totali, ovvero considerando anche la dimensione della *ghost area*. All'interno di questo kernel, invece di inizializzare solamente le celle "fantasma", il valore scelto viene associato a tutti gli elementi dei canali, così da evitare di inserire condizioni in più oltre a quelle sulla dimensione dell'immagine e far lavorare tutti i *threads* creati eccetto quelli in più aggiunti per riempire i blocchi in caso la dimensione della matrice non sia multipla della dimensione del blocco. Al termine dell'inizializzazione dei canali usati come output è possibile iniziare l'esecuzione delle operazioni di erosione e di dilatazione, eseguite rispettivamente dai kernel `Erosion(…)` e `Dilation(…)`; per entrambi i parametri passati sono `input` e `output` contenenti il canale su cui

eseguire l'operazione e quello che dovrà contenere il risultato, `width` ed `height` per la larghezza e l'altezza dell'immagine escluse le *ghost cells*, `structWidth` e `structHeight` per quelle dell'elemento strutturante ed `offset` ed `offCount` rispettivamente per gli *offsets* e il numero di essi. In questi kernel, invece di controllare semplicemente che le coordinate del *thread* non superino le dimensioni effettive dell'immagine, viene invece verificato che esse non vadano ad indicare punti della *ghost area* perché erosione e dilatazione vanno calcolate solamente per i pixel reali dell'immagine, mentre le *ghost cells* servono solo come appoggio. Infine, l'immagine risultato deve essere ricomposta e per far ciò viene eseguito il kernel `ComposeImage (uint8_t* image, uint8_t* red, uint8_t* green, uint8_t* blue, int structWidth, int structHeight, int width, int height)`, in cui `image` conterrà l'immagine risultante, mentre gli altri parametri sono analoghi a quelli di `SplitChannels(…)`. Come in quest'ultimo, anche in `ComposeImage(…)` l'indice con cui vengono acceduti i valori dei canali viene calcolato come la variabile `index` del Frammento 27 a pagina 73, mentre quello per `image` viene calcolato come `i`, visualizzabile nello stesso frammento, eccetto il controllo sulla *ghost area*. La condizione che sceglie quali *threads* mandare in esecuzione, infatti, è analoga a quella presente nei kernel `Erosion(…)` e `Dilation(…)`, perché le *ghost cells* non devono comparire nel risultato, quindi la variabile `i` sarà sempre un indice valido per l'immagine.

4.3. Analisi delle prestazioni

Come per l'algoritmo descritto nel capitolo 3, anche le implementazioni delle operazioni di morfologia matematica sono state analizzate calcolando i tempi di esecuzione sull'hardware descritto nella sezione 3.3. Come già detto in **4.2.2 Versione parallela con OpenMP**, questo tipo di implementazione parallela può essere eseguita in modi diversi: utilizzando le sezioni, oppure parallelizzando i cicli for dei vari metodi. Dato che è possibile parallelizzare in sezioni sia l'inizializzazione dei canali che le operazioni da svolgere su di essi, per verificare quale metodo di parallelizzazione porta a prestazioni migliori sono stati presi i tempi per quattro versioni diverse: in Tabella 4 vengono riportati i

tempi per la prima, che fa uso delle sezioni in tutti i punti in cui è possibile utilizzarle.

p	1	2	3	4	5	6	7	8	9	10
	1,956	1,303	0,67	0,673	0,668	0,671	0,671	0,67	0,671	0,671
	1,95	1,303	0,671	0,672	0,669	0,674	0,673	0,672	0,672	0,673
	1,948	1,305	0,671	0,672	0,672	0,672	0,672	0,672	0,679	0,675
	1,949	1,304	0,672	0,671	0,671	0,671	0,671	0,674	0,67	0,67
	1,951	1,306	0,671	0,673	0,67	0,67	0,67	0,67	0,671	0,672
Media	1,9508	1,3042	0,671	0,6722	0,67	0,6716	0,6714	0,6716	0,6726	0,6722
S(p) v1	1	1,495783	2,907303	2,902112	2,911642	2,904705	2,90557	2,904705	2,900387	2,902112
E(p) v1	1	0,747891	0,969101	0,725528	0,582328	0,484118	0,415081	0,363088	0,322265	0,290211

Tabella 4 - Tempi calcolati per la prima versione di OpenMP con SE:11x11 su un'immagine 1920x2560

Nella seconda, invece, i cui tempi sono riportati in Tabella 5, le sezioni vengono utilizzate solamente per parallelizzare l'inizializzazione dei canali, mentre nella terza esse vengono usate solo per l'esecuzione delle operazioni (Tabella 6).

p	1	2	3	4	5	6	7	8	9	10
	2,073	1,049	0,714	0,542	0,459	0,378	0,606	0,542	0,485	0,442
	2,07	1,049	0,713	0,542	0,439	0,383	0,535	0,511	0,466	0,473
	2,07	1,051	0,712	0,542	0,439	0,389	0,544	0,549	0,485	0,475
	2,071	1,05	0,715	0,543	0,439	0,384	0,616	0,509	0,483	0,443
	2,069	1,051	0,712	0,542	0,439	0,379	0,606	0,542	0,487	0,477
Media	2,0706	1,05	0,7132	0,5422	0,443	0,3826	0,5814	0,5306	0,4812	0,462
S(p) v2	1	1,972	2,903253	3,818886	4,674041	5,411918	3,561404	3,902375	4,302993	4,481818
E(p) v2	1	0,986	0,967751	0,954722	0,934808	0,901986	0,508772	0,487797	0,47811	0,448182

Tabella 5 - Tempi calcolati per la seconda versione di OpenMP con SE:11x11 su un'immagine 1920x2560

p	1	2	3	4	5	6	7	8	9	10
	1,95	1,296	0,661	0,659	0,659	0,658	0,66	0,661	0,659	0,659
	1,953	1,296	0,663	0,66	0,66	0,66	0,662	0,663	0,664	0,66
	1,947	1,301	0,662	0,659	0,662	0,66	0,664	0,663	0,663	0,665
	1,956	1,301	0,661	0,66	0,659	0,658	0,661	0,663	0,66	0,66
	1,954	1,304	0,666	0,658	0,658	0,66	0,662	0,662	0,658	0,661
Media	1,952	1,2996	0,6626	0,6592	0,6596	0,6592	0,6618	0,6624	0,6608	0,661
S(p) v3	1	1,502001	2,94597	2,961165	2,959369	2,961165	2,949532	2,94686	2,953995	2,953101
E(p) v3	1	0,751	0,98199	0,740291	0,591874	0,493528	0,421362	0,368357	0,328222	0,29531

Tabella 6 - Tempi calcolati per la terza versione di OpenMP con SE:11x11 su un'immagine 1920x2560

Infine, l'ultima versione, a cui fanno riferimento i tempi della Tabella 7, utilizza solo il parallelismo dei cicli.

p	1	2	3	4	5	6	7	8	9	10
	2,076	1,045	0,704	0,534	0,43	0,369	0,605	0,532	0,474	0,471
	2,071	1,047	0,707	0,534	0,429	0,369	0,498	0,505	0,479	0,439
	2,074	1,05	0,704	0,531	0,428	0,371	0,497	0,498	0,46	0,447
	2,071	1,048	0,705	0,532	0,432	0,374	0,496	0,502	0,473	0,428
	2,066	1,045	0,707	0,534	0,428	0,371	0,593	0,499	0,452	0,46
Media	2,0716	1,047	0,7054	0,533	0,4294	0,3708	0,5378	0,5072	0,4676	0,449
S(p) v4	1	1,978606	2,936773	3,886679	4,824406	5,586839	3,85199	4,084385	4,430282	4,613808
E(p) v4	1	0,989303	0,978924	0,97167	0,964881	0,93114	0,550284	0,510548	0,492254	0,461381

Tabella 7 - Tempi calcolati per la quarta versione di OpenMP con SE:11x11 su un'immagine 1920x2560

I tempi mostrati nelle quattro tabelle precedenti sono stati presi eseguendo l'operazione di apertura su un'immagine di dimensione $1920x2560$ utilizzando un elemento strutturante $11x11$, poi, dopo aver calcolato le medie, tali valori sono stati usati per creare il grafico dello *speedup* seguente (Grafico 6).

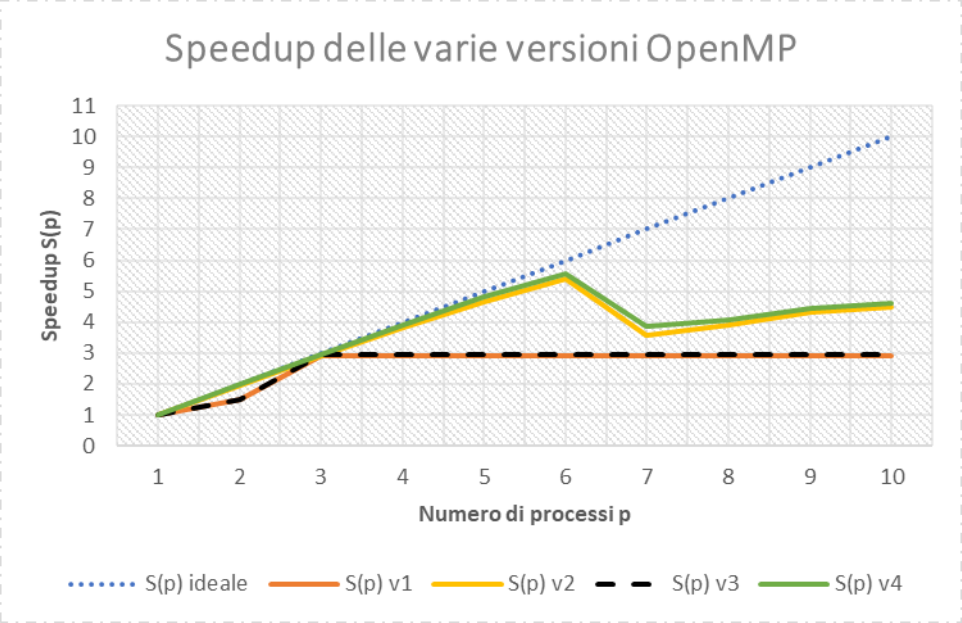

Grafico 6 - Grafico dello speedup delle varie versioni di OpenMP con SE:11x11 su un'immagine 1920x2060

In esso vengono mostrati con una linea blu tratteggiata i valori dello *speedup* ideale, mentre una linea continua arancione mostra i tempi della prima versione, una gialla per la seconda, una nera per la terza e quella verde per l'ultima. È stato deciso di mostrare la linea nera della terza versione a tratteggi perché, come si può osservare dal grafico e dai valori di Tabella 4 e Tabella 6, lo *speedup* della prima e della terza

versione sono molto simili tra loro, tanto che le due linee nel grafico sono sovrapposte: se anche la linea della terza versione fosse stata continua, quella della prima non sarebbe stata visibile. Dal grafico, inoltre, si può notare che per le versioni 1 e 3 lo *speedup* massimo si raggiunge con $p = 3$, poi per i successivi valori di p resta costante, proseguendo in linea retta; le versioni 2 e 4, invece, hanno un picco di *speedup* con $p = 6$, non più raggiungibile aumentando il numero di processi. Si nota, inoltre, che mentre per le versioni 2 e 4 lo *speedup* è quasi lineare fino al picco, per le altre due versioni lo *speedup* ottenuto con due *threads* si discosta dal suo valore ideale per poi riavvicinarsi con $p = 3$. Queste differenze sono date dal metodo di implementazione delle sezioni: esse, infatti, vengono ognuna presa in carico da un singolo *thread* che eseguirà in maniera sequenziale il contenuto della sua sezione. Durante la fase di inizializzazione dei canali le sezioni create sono due, una per eseguire il metodo `SplitChannels(…)` e una per `FillGhostCells(…)`, mentre per l'esecuzione delle operazioni sono una per ogni canale, ovvero tre. Eseguire la versione 1 con soli due *threads*, significa che, durante l'inizializzazione ognuno di essi si occupa di una sezione, mentre per quelle successive due vengono eseguite contemporaneamente, mentre la terza aspetta il primo *thread* che si libera dell'esecuzione di una delle altre: questo fa sì che due canali vengano eseguiti sequenzialmente, allungando quindi i tempi di esecuzione. Per la terza versione il discorso è analogo perché il collo di bottiglia non è dato dall'inizializzazione, che in questo caso è eseguita parallelizzando i cicli, ma dall'esecuzione dell'operazioni in sezioni. Queste considerazioni sono anche il motivo per cui lo *speedup* resta costante come quello calcolato con tre *threads* anche aumentandone il numero: dato che i canali sono tre, solamente tre dei *threads* creati eseguiranno una sezione, quindi crearne in più non fa diminuire i tempi di esecuzione. A differenza di queste due versioni, la seconda si comporta in maniera diversa perché in essa le sezioni vengono usate solo per l'inizializzazione dei canali, mentre il carico di lavoro maggiore, quello per l'esecuzione delle operazioni morfologiche, viene eseguito parallelizzando i cicli; tale versione, quindi, non è soggetta al calo di *speedup* se eseguita con soli due *threads* perché ogni sua sezione viene eseguita in contemporanea, essendo solo due, mentre le operazioni sui vari canali vengono svolte una di seguito all'altra ma

parallelizzando le varie iterazioni che le compongono. I tempi di esecuzione di ogni operazione risultano in questo modo più brevi, dunque anche eseguendole sequenzialmente si risparmia tempo rispetto alla loro esecuzione seriale; inoltre, non avendo sezioni che vincolano il numero di *threads* da utilizzare, parallelizzando i cicli le loro iterazioni possono essere suddivise anche tra più *threads*, così da poter raggiungere lo *speedup* massimo sfruttando al meglio i vari processori della CPU. Mentre le versioni 1 e 3 hanno *speedup* pressoché identici, tra la versione 2 e la 4 è presente una lieve differenza che fa risultare più prestante l'ultima: questo è dato dal fatto che, benché la maggior parte del carico di lavoro sia svolto allo stesso modo, nella versione 2 sono ancora presenti le sezioni per l'inizializzazione dei canali, che nella 4 sono state sostituite. Anche se l'esecuzione dei metodi contenuti in queste due ultime sezioni non influisce di molto sui tempi di esecuzione totali, durante la loro esecuzione nella seconda versione vengono sempre utilizzati solamente due *threads*, mentre parallelizzandoli tramite `#pragma omp for` tutti i *threads* creati eseguirebbero una porzione dei cicli, riducendo anche per questi metodi i loro tempi di esecuzione e migliorando dunque i tempi totali. Per un ulteriore studio sulle varie versioni, nel seguente Grafico 7 viene riportata la loro scalabilità: come prima i valori ideali sono mostrati con una linea tratteggiata blu, mentre per le versioni 1, 2 e 4 sono utilizzate rispettivamente una linea continua arancione, una gialla e una verde, mentre per la terza versione la linea è nera tratteggiata per non coprire completamente quella della prima implementazione, del tutto sovrapposta ad essa.

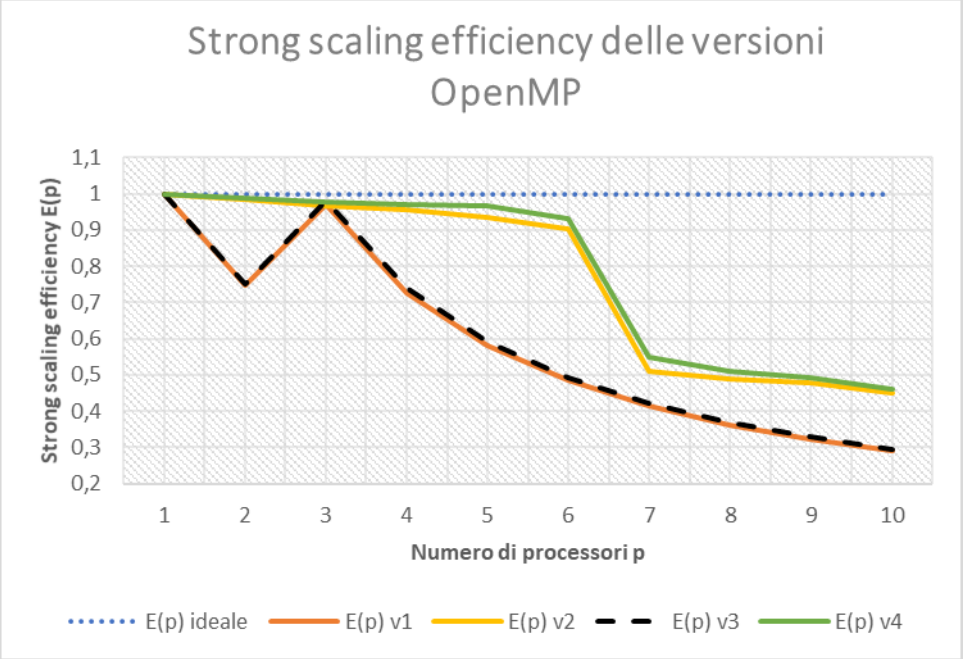

Grafico 7 - Strong scaling efficiency delle versioni OpenMP con SE:11x11 su un'immagine 1920x2560

Da questo grafico è ancora più visibile la differenza tra le versioni che utilizzano le sezioni per l'esecuzione dei vari canali e quelle che invece parallelizzano i cicli: per le versioni 1 e 3, infatti, è molto visibile il picco dato da $p = 2$, che rende ancora più evidente quanto sia poco conveniente utilizzare solo due *threads* per implementazioni di questo tipo; superato $p = 3$, inoltre, si nota una curva discendente data dal fatto che lo *speedup* resta costante all'aumentare del numero di processi in uso. Per le versioni 2 e 4, come era da prevedere, la *strong scaling efficiency* resta quasi costante fino a $p = 6$, perché lo *speedup* è quasi lineare per questo numero di processi, mentre poi inizia a calare poiché è stato superato il numero di core fisici della CPU che possono essere usati; come nel grafico dello *speedup*, anche qui è evidente che l'ultima versione proposta è quella più efficiente.

Per quanto riguarda la versione CUDA, dato che non è possibile calcolare *speedup* ed efficienza, le sue prestazioni vengono analizzate mettendo a confronto i suoi tempi di esecuzione con quelli ottenuti dalla quarta versione OpenMP eseguita prima con un *thread*, poi con sei. Nel Grafico 8 vengono riportati i tempi medi di esecuzione dell'operazione di apertura su un'immagine $1920x2560$.

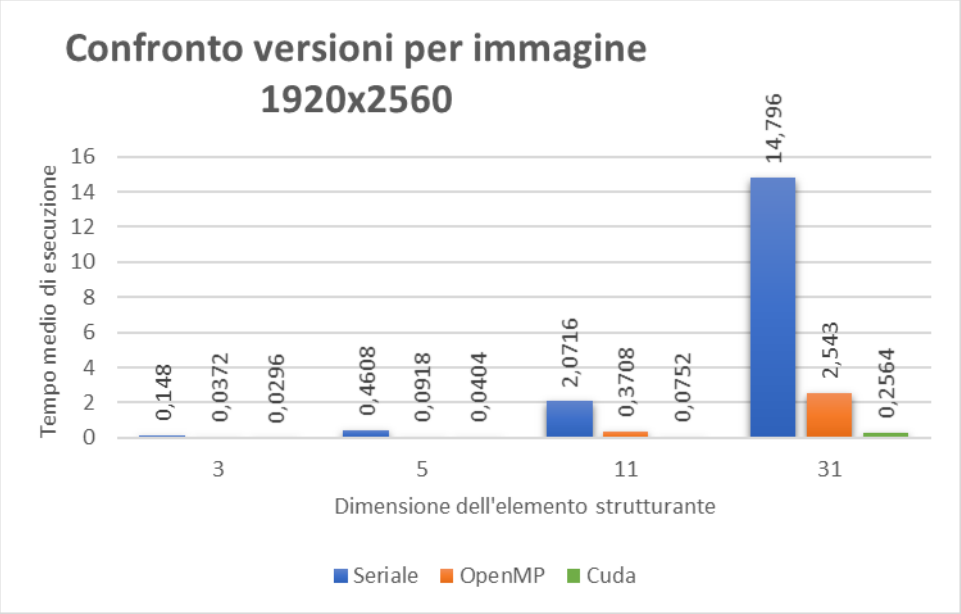

Grafico 8 - Grafico di confronto tra le versioni seriale, OpenMP e CUDA su un'immagine 1920x2560

In questo grafico gli istogrammi blu indicano i tempi medi delle esecuzioni seriali, quelli arancioni rappresentano le esecuzioni della versione OpenMP, mentre quelli verdi quelle svolte con CUDA: è evidente che la versione parallelizzata con CUDA risulta essere molto più efficiente di quella seriale e, anche se con elementi strutturanti di piccole dimensioni l'esecuzione OpenMP ha tempi molto simili a quelli di CUDA, i tempi di quest'ultima implementazione crescono molto più lentamente rispetto a quelli di OpenMP. Per verificare tale ipotesi vengono mostrati altri due grafici in cui vengono confrontati i tempi medi di esecuzione dell'operazione di apertura su altre due immagini, una più piccola, di dimensione $640x480$, e una più grande, di dimensione $3264x2448$. Come per il Grafico 8, il seguente Grafico 9, riportante i dati calcolati per l'immagine piccola, contiene istogrammi blu per i tempi seriali, arancioni per quelli di OpenMP e verdi per CUDA. Da esso si nota che utilizzando l'elemento strutturante di dimensione $3x3$, anche se la differenza è poca, i tempi migliori sono dati dalla versione OpenMP; ciò può essere causato dal fatto che mandare in esecuzione i vari *threads* della GPU produce *overhead*, così come trasferire memoria dall'*host* al *device* e viceversa. Se la matrice è piccola, quindi, l'*overhead* ha un peso maggiore sul calcolo del tempo di esecuzione, aumentandolo.

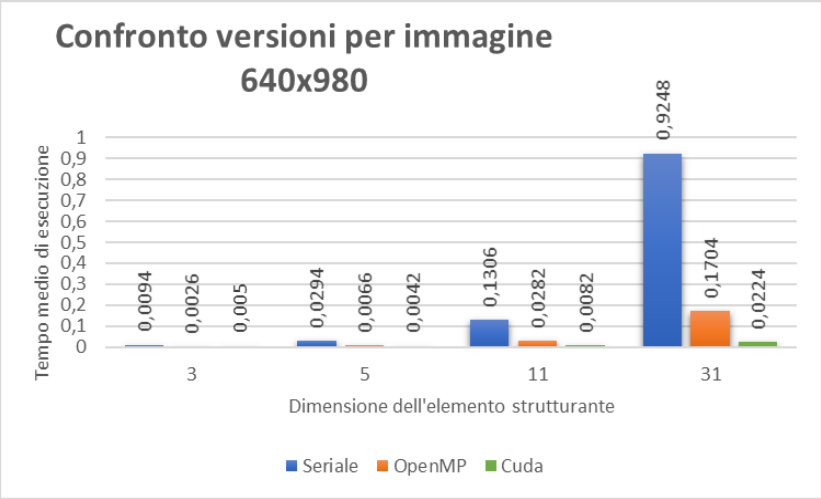

Grafico 9 - Grafico di confronto tra le versioni seriale, OpenMP e CUDA su un'immagine 640x480

I tempi di esecuzione ottenuti eseguendo operazioni morfologiche sull'immagine più grande, invece, vengono riportati nel seguente Grafico 10. In esso, per ogni dimensione dell'elemento strutturante, la versione CUDA risulta essere la migliore, come per il Grafico 8.

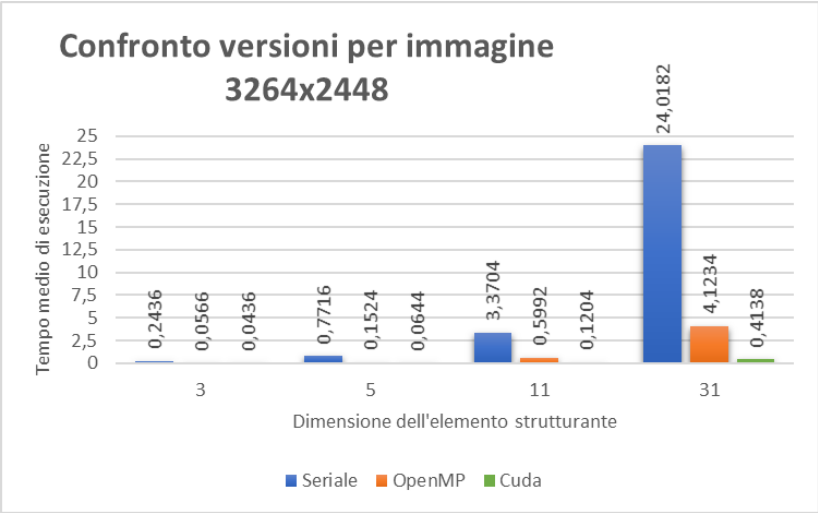

Grafico 10 - Grafico di confronto tra le versioni seriale, OpenMP e CUDA su un'immagine 3264x2448

Tutte le analisi eseguite, come già accennato, sono riferite solamente all'operazione morfologica di apertura, tuttavia tutto ciò che è stato detto vale anche per la chiusura poiché le operazioni da eseguire sono le stesse svolte per l'apertura, semplicemente disposte in ordine diverso: i tempi di esecuzione quindi restano gli stessi, perciò anche lo *speedup* e la scalabilità della versione OpenMP non cambiano.

5. Conclusioni

In questa tesi sono stati studiati due diversi algoritmi di elaborazione di immagini: il primo è l'algoritmo Diamond-Square, con cui è possibile creare texture proceduralmente. Per questo algoritmo le varie analisi hanno portato a preferire l'implementazione parallela svolta con OpenMP perché l'utilizzo di CUDA risulta meno efficiente, forse anche per il fatto che ad ogni iterazione il numero di *threads* creati è molto piccolo, eccetto all'ultimo *square step*, con cui viene inizializzata la metà degli elementi della matrice. Dato che la creazione di *threads* crea *overhead*, utilizzare piccoli blocchi fa sì che esso risulti il carico di lavoro maggiore, rendendo quindi i tempi più elevati. Aumentando la dimensione dell'immagine, tuttavia, si nota che i tempi ottenuti dall'esecuzione CUDA crescono più lentamente di quelli di OpenMP, quindi ipoteticamente, per domini infinitamente grandi CUDA è il metodo di implementazione migliore. Per quanto riguarda il secondo algoritmo, invece, le varie implementazioni permettono di eseguire due diverse operazioni di morfologia matematica, apertura e chiusura, a loro volta realizzabili alternando l'esecuzione di altre due operazioni morfologiche, erosione e dilatazione. Tali operazioni solitamente hanno come dominio immagini binarie, i cui pixel possono assumere solamente due valori, 0 per il nero e 255 per il bianco. Esiste, tuttavia, una generalizzazione della morfologia matematica che permette di applicarla anche su immagini in scala di grigio, quindi in questa tesi si è deciso di applicare tale generalizzazione sui vari canali di un'immagine a colori, provando così ad eseguire queste operazioni anche su immagini RGB. Come ci si aspettava le operazioni di apertura rendono l'immagine più scura, mentre quelle di chiusura la rendono più chiara: questo comportamento è dato dal fatto per l'apertura la prima

operazione da svolgere, quindi quella che influenza di più il risultato finale, è l'erosione che calcola ogni pixel assegnandogli il minimo del suo interno. Tale pixel, dunque riceverà un colore più scuro, mentre quelli calcolati con la dilatazione saranno più chiari perché ad essi viene assegnato il massimo dell'intorno; dato che la dilatazione è la prima operazione da svolgere per la chiusura, quindi, è questo il motivo per cui il risultato di essa è un'immagine più chiara. Per questo tipo di operazioni sono state proposte diverse versioni OpenMP, il cui codice è riportato commentato anche per le versioni meno efficienti; la versione parallela migliore, tuttavia, risulta essere quella implementata con CUDA. Il codice del progetto Unreal Engine e delle librerie create per eseguire CUDA è stato inserito in un repository GitHub al link https://github.com/Celinn/HPCImageProcessing.git in cui è presente anche il file README in cui vengono riportati i requisiti per la compilazione e l'esecuzione di tale progetto. Infine, all'interno della cartella InputImage tra le cartelle del progetto Unreal Engine sono presenti le tre immagini utilizzate per l'analisi delle versioni parallele delle operazioni di morfologia matematica: una è una fotografia di una rosa, mentre le altre sono immagini di PixelJoy WebDesign, scaricate in varie dimensioni dal sito Pixabay [4].

[4] Sito al link: https://pixabay.com/it/photos/cuscino-puntino-nero-bianco-nizza-120763

Bibliografia

[1] M. J. Flynn, «Very high-speed computing systems,» *Proceedings of the IEEE,* vol. 54, n. 12, 1966.

[2] J. von Neumann, «First draft of a report on the EDVAC,» *IEEE Annals of the History of Computing,* vol. 15, n. 4, 1993.

[3] «Intel intrinsics guide,» Intel, [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/#. [Consultato il giorno 25 Febbraio 2020].

[4] K. O. W. Group, «The OpenCL™ Specification,» 19 Luglio 2019. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf. [Consultato il giorno 3 Febbraio 2020].

[5] «NVCC,» NVIDIA, 28 Novembre 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html. [Consultato il giorno 1 Febbraio 2020].

[6] «CUDA C++ Programming Guide,» NVIDIA, 28 Novembre 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html. [Consultato il giorno 1 Febbraio 2020].

[7] P. Pacheco, An introduction to parallel programming, Morgan Kaufmann, 2011.

[8] R. C. Gonzales e R. E. Woods, Digital Image Processing, Addison-Wesley Pub (Sd), 2001.

[9] «Blueprints visual scripting,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/Engine/Blueprints/index.html. [Consultato il giorno 31 Gennaio 2020].

[10] «Downloading Unreal Engine Source Code,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/GettingStarted/DownloadingUnrealEngine/index.html. [Consultato il giorno 31 Gennaio 2020].

[11] «Building Unreal Engine from Source,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/Programming/Development/BuildingUnrealEngine/index.html. [Consultato il giorno 31 Gennaio 2020].

[12] «Linking Static Libraries Using The Build System,» Epic Games, [Online]. Available: https://wiki.unrealengine.com/Linking_Static_Libraries_Using_The_Build_System. [Consultato il giorno 31 Gennaio 2020].

[13] «Widget Blueprints,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/Engine/UMG/UserGuide/WidgetBlueprints/index.html. [Consultato il giorno 5 Febbraio 2020].

[14] «Quit Game,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/BlueprintAPI/Game/QuitGame/index.html. [Consultato il giorno 5 Febbraio 2020].

[15] «Creating widgets,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/Engine/UMG/UserGuide/CreatingWidgets/index.html. [Consultato il giorno 5 Febbraio 2020].

[16] «Set Visibility,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/BlueprintAPI/Widget/SetVisibility/index.html. [Consultato il giorno 5 Febbraio 2020].

[17] «Remove From Parent,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/BlueprintAPI/Widget/RemovefromParent/index.html. [Consultato il giorno 5 Febbraio 2020].
[18] «Blueprint function libraries,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/Programming/BlueprintFunctionLibraries/index.html. [Consultato il giorno 9 Febbraio 2020].
[19] «Level Blueprint,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/Engine/Blueprints/UserGuide/Types/LevelBlueprint/index.html. [Consultato il giorno 9 Febbraio 2020].
[20] «UFunctions,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/Programming/UnrealArchitecture/Reference/Functions/index.html. [Consultato il giorno 9 Febbraio 2020].
[21] A. Fourier, D. Fussell e L. Carpenter, «Computer rendering of stochastic models,» *Comminication of the ACM*, vol. 25, n. 6, 1982.
[22] «POV-Ray: Documentation: 2.4.1.5 Height Field,» [Online]. Available: http://www.povray.org/documentation/view/3.6.1/279/. [Consultato il giorno 9 Febbraio 2020].
[23] G. S. P. Miller, «The definition and rendering of terrain maps,» *ACM SIGGRAPH Computer Graphics*, pp. 39-48, 1986.
[24] «Funzionalità libreria CRT,» Microsoft, [Online]. Available: https://docs.microsoft.com/it-it/cpp/c-runtime-library/crt-library-features?view=vs-2019. [Consultato il giorno 13 Febbraio 2020].
[25] «srand,» Microsoft, [Online]. Available: https://docs.microsoft.com/it-it/cpp/c-runtime-library/reference/srand?view=vs-2019. [Consultato il giorno 13 Febbraio 2020].
[26] «time, _time32, _time64,» Microsoft, [Online]. Available: https://docs.microsoft.com/it-it/cpp/c-runtime-library/reference/time-time32-time64?view=vs-2019. [Consultato il giorno 13 Febbraio 2020].
[27] «4.1. SIMT Architecture,» NVIDIA, 28 Novembre 2019. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture. [Consultato il giorno 15 Febbraio 2020].
[28] «CUDA Toolkit 4.2 CURAND Guide,» NVIDIA, Marzo 2012. [Online]. Available: https://developer.download.nvidia.com/compute/DevZone/docs/html/CUDALibraries/doc/CURAND_Library.pdf. [Consultato il giorno 15 Febbraio 2020].
[29] I. L. Dalal, D. Stefan e J. Harwayne-Gidansky, «Low Discrepancy Sequences for Monte Carlo Simulations on Reconfigurable Platforms,» [Online]. Available: https://cseweb.ucsd.edu/~dstefan/pubs/dalal:2008:low.pdf. [Consultato il giorno 15 Febbraio 2020].
[30] G. Marsaglia, «Xorshift RNGs,» *Journal of Statistical Software*, vol. 8, n. 14, 2003.
[31] G. M. Amdahl, «Validity of the single processor approach to achieving large scale computing capabilities,» *Proceedings of the American Federation of Information Processing Societies Conference*, vol. 30, n. 2, pp. 483-485, 1967.
[32] R. Cappelli, «Morfologia matematica,» [Online]. Available: http://bias.csr.unibo.it/fei/Dispense/5%20-%20FEI%20-%20Morfologia%20Matematica.pdf. [Consultato il giorno 15 Febbraio 2020].

[33] «FImage,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/API/Runtime/ImageCore/FImage/index.html. [Consultato il giorno 19 Febbraio 2020].
[34] «FColor,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/API/Runtime/Core/Math/FColor/index.html. [Consultato il giorno 19 Febbraio 2020].
[35] «TArray,» Epic Games, [Online]. Available: https://docs.unrealengine.com/en-US/API/Runtime/Core/Containers/TArray/index.html. [Consultato il giorno 19 Febbraio 2020].