Alma Mater Studiorum · Università di Bologna

Corso di Laurea in Ingegneria e Scienze Informatiche

PARALLELIZZAZIONE DELLA GENERAZIONE PROCEDURALE DI UN MONDO 3D A BLOCCHI

Elaborato in
HIGH PERFORMANCE COMPUTING

Relatore:
Prof. MORENO MARZOLLA

Presentata da:
ALIN ȘTEFAN BORDEIANU

Sessione III
Anno Accademico 2023-2024

"Uno non ha che dichiararsi libero, ed ecco che in quello stesso istante si sente limitato. Abbia solo il coraggio di dichiarasi limitato, ed eccolo libero."

— Goethe

Indice

Introduzione

In questa tesi si discuterà l'implementazione di un algoritmo di generazione procedurale di un mondo 3D a blocchi ispirato al videogioco Minecraft. L'algoritmo verrà realizzato sfruttando tecniche di *high performance computing* che consentono di parallelizzarne l'esecuzione mediante una GPU (*Graphics Processing Unit*).

Il codice verrà realizzato a partire da una versione del gioco esistente e realizzata in codice C esclusivamente seriale, ossia pensato per essere eseguito su una sola CPU (*Central Processing Unit*). Tale versione è disponibile in rete.

Le motivazioni alla base della scelta di questo progetto derivano da alcune considerazioni sull'attualità del mondo dei videogiochi, in cui si nota una tendenza ad abusare delle potenzialità sempre maggiori dell'hardware a discapito di chi non si può permettere macchine e componenti di ultima generazione. Questa tesi è un tentativo esplicito di mettere in campo le attuali conoscenze in ambito di calcolo parallelo al servizio di una funzionalità che molti giochi presentano, ossia la generazione di un mondo infinito in maniera dinamica a seconda della posizione del giocatore. I risultati raggiunti tuttavia non si limiteranno ad avere applicazioni esclusivamente in ambito videoludico, in quanto i problemi affrontati durante il suo svolgimento hanno validità più generale per tutti i campi in cui queste ottimizzazioni sono applicabili, come ad esempio le simulazioni e la computer grafica.

Saranno presenti quindi, oltre alle valutazioni effettive delle prestazioni, alla descrizione dei problemi affrontati e delle soluzioni adottate, anche alcune osservazioni pratiche su un possibile approccio per l'inserimento di porzioni di codice parallelizzato su GPU in progetti esistenti e non congegnati apposta per l'applicazione di questa tecnica di ottimizzazione. Più nello specifico, si discuteranno alcuni dettagli utili relativi al debugging di applicazioni CUDA, alla creazione di file di automazione della compilazione in ambito Unix che compilino codice in linguaggi diversi, ad un algoritmo di generazione procedurale, e allo studio di codice preesistente nel tentativo di individuare opportunità di ottimizzazione tramite calcolo parallelo.

Capitolo 1

Gli algoritmi per la generazione procedurale

In questo capitolo verranno fornite informazioni di carattere generale su alcuni algoritmi di generazione procedurale, con particolare attenzione a quelli impiegati nei videogiochi.

1.1 Definizione

Quando si parla di *generazione* all'interno di un videogioco ci si riferisce alla creazione e utilizzo di elementi che costituiscono l'ambiente di gioco (mappa di gioco, in cui possono essere posizionati ostacoli, elementi di fauna e flora, oggetti, edifici, elementi naturali) al momento dell'esecuzione. Questo processo è diverso dall'utilizzare una risorsa salvata su disco, caso in cui uno sviluppatore deve aver predisposto e deciso la forma di tale contenuto in tutti i suoi dettagli. La generazione può consentire un'alta rigiocabilità e una fonte di novità continua.

Qualsiasi forma di generazione dei contenuti governata da un algoritmo (o procedura) è detta *generazione procedurale*. A differenza della generazione casuale, quella procedurale non implica in sé casualità di alcun tipo. Fintanto che gli input di partenza e l'algoritmo usato restano costanti, l'output prodotto sarà sempre lo stesso. Nel caso della generazione casuale, ciò che chiamiamo casualità è in realtà spesso un prodotto della variazione degli input dati ad un algoritmo. La vera casualità si ottiene solamente con hardware specifico, dal momento che i computer sono macchine deterministiche. Per i videogiochi, un normale PC con sistemi di generazione di numeri pseudo-casuali è più che sufficiente per produrre i risultati desiderati [5].

1.2 Esempi di algoritmi di generazione procedurale

Data la vastità del termine e degli ambiti a cui la generazione procedurale può essere applicata, non esiste al momento uno standard utilizzabile indiscriminatamente in ogni situazione. Ciò che accade nella pratica è che ogni progetto adatta o crea le proprie tecniche in base alle proprie esigenze, spesso combinando più di un approccio. Introduciamo alcune strategie comuni, riservando particolare attenzione alla tecnica del *Perlin Noise* che viene sfruttata nel caso in analisi.

1.2.1 Automi cellulari

Gli automi cellulari sono una classe di sistemi matematici deterministici, discreti spazialmente e temporalmente. Sono modelli che prototipano sistemi complessi, e consistono in un gran numero di unità semplici (cellule o celle) che interagiscono localmente ed evolvono nel tempo secondo regole precise. Dal momento della loro introduzione da parte di von Neumann all'inizio degli anni '50 del secolo scorso hanno suscitato enorme interesse nel campo della ricerca per la loro capacità di far emergere una grande complessità di schemi comportamentali a partire da regole semplici.

Elementi ricorrenti negli automi cellulari sono:

- *Griglia discreta di celle*: si sfrutta una griglia 1D, 2D o 3D di celle discreta, ossia in cui ciascuna cella assume una posizione data da coordinate intere;
- *Omogeneità*: tutte le cellule sono equivalenti;
- *Stati discreti*: tutte le celle si trovano in uno stato, e gli stati possibili sono in numero finito;
- *Interazioni locali*: tutte le celle interagiscono solo con le celle nelle proprie vicinanze;
- *Dinamiche discrete*: ad ogni unità discreta di tempo trascorso, ogni cella aggiorna il proprio stato in base alle regole definite che tengono conto degli stati delle celle vicine [9].

Un esempio classico è l'automa cellulare di Conway, conosciuto come "The Game of Life" [10].

A seconda delle regole adottate, modelli di generazione procedurale basati su automi cellulari potrebbero realizzare caverne o stanze sotterranee dall'aspetto organico e prive di ostacoli formati da una sola unità. Partendo da un insieme iniziale di punti distribuiti casualmente su una griglia, si può lanciare la simulazione dell'automa cellulare per un dato numero di generazioni e decidere ad esempio che ogni cella su cui si è trovata una cellula nello stato "viva" sia una casella libera su cui il giocatore si possa spostare.

1.2.2 Algoritmi evolutivi

Gli algoritmi evolutivi prendono a loro volta ispirazione dal mondo biologico e a partire da una popolazione iniziale di elementi generici si avvia un processo di evoluzione diviso in più generazioni. Ad ogni generazione, gli individui della popolazione possono mutare il proprio stato, riprodursi trasmettendo i propri "geni" opportunamente definiti ai figli (i quali possono andare incontro a fenomeni di ricombinazione che li dotano di caratteristiche diverse da quelle dei genitori) e subire la selezione al termine della generazione.

Il processo di selezione, ispirato alla selezione naturale, consente soltanto all'insieme di individui che posseggono caratteristiche "soddisfacenti" di continuare a sopravvivere. La loro sopravvivenza è determinata da una funzione di *idoneità* (in inglese *fitness*) che deve essere opportunamente definita.

Una mappa di gioco potrebbe essere creata a partire da un insieme di stanze collegate da passaggi e porte (eventualmente chiuse a chiave), e nuove stanze possono essere aggiunte mediante i meccanismi evolutivi citati in precedenza. Più mappe candidate possono essere realizzate, e soltanto quelle che soddisfano i criteri della funzione di idoneità vengono prese in considerazione per il gioco effettivo [16].

1.2.3 Cammini casuali (*random walk*)

Scelto un punto di partenza sulla mappa, si effettua ad ogni iterazione un passo in una direzione casuale. Se sul cammino così individuato si piazzano stanze, ostacoli naturali o altri tipi di contenuto, si assicura l'esistenza di un percorso che li colleghi tutti. Più formalmente, in ambito matematico un cammino casuale è definito come la serie di passi effettuati su una griglia d-dimensionale intera $\mathbb{Z}^d$. Ogni passo, o salto, viene effettuato secondo una distribuzione di probabilità che dipende dalle fluttuazioni o dai vincoli imposti al sistema (ad esempio, si potrebbe assegnare una maggiore probabilità di accadere a passi che portano verso il centro della griglia) [4].

1.2.4 Diagrammi di Voronoi

I diagrammi di Voronoi sono una forma di tassellatura dello spazio. Si parte da un piano (o una regione tridimensionale) su cui sono definiti dei punti di partenza arbitrari, spesso determinati casualmente. Ad ogni punto P viene poi assegnata una partizione del piano, il cui perimetro è formato da tutti i punti più vicini a P che ad ogni altro punto. Questo metodo garantisce una tassellatura completa dello spazio, ed è solitamente applicato al caso bidimensionale.

Applicazioni interessanti di questo metodo possono riscontrarsi nella creazione di caverne contenenti stanze con certe caratteristiche desiderabili per renderle appaganti da esplorare, ad esempio focalizzando la generazione su un insieme di punti iniziali di

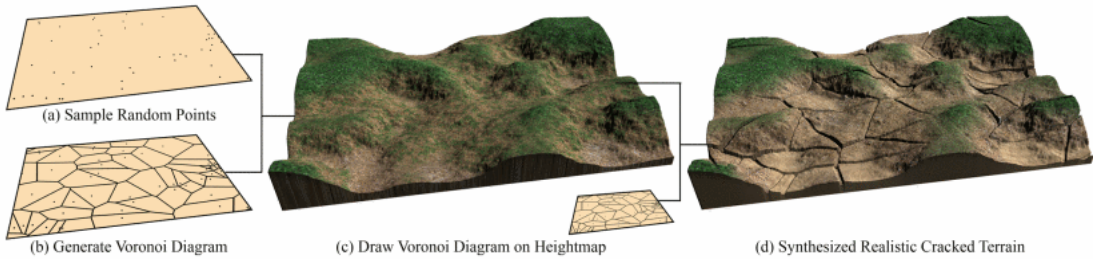

Figura 1.1: Applicazione di diagrammi di Voronoi per la generazione di terreni aridi. Fonte: [12].

interesse e costruendo gli antri della caverna attorno a questi [18]. Un altro esempio è usare la forma settorizzata di questi diagrammi per rappresentare le scanalature dei terreni in secca [12], come mostrato nella Figura 1.1.

1.2.5 Mappe altimetriche (*height map*)

Più che una tecnica di generazione procedurale a sé stante, le mappe altimetriche o *height map* sono delle strutture dati che rappresentano l'altezza (asse verticale Z) in funzione delle coordinate del piano orizzontale (X, Y). Un modo di rappresentarle è assegnare un byte (256 possibili valori) per ciascuna cella della griglia orizzontale e colorarla sfruttando una scala di grigi in cui il bianco rappresenta i punti alla massima altezza e il nero quelli all'altezza minima. La mappa così creata può essere poi data come input ai vari algoritmi che si occupano di piazzare gli elementi naturali del terreno a seconda della distribuzione di altezze.

Lo svantaggio di questo metodo è che non consente di per sé di creare caverne o elementi sospesi sopra ad altri, in quanto non vengono memorizzate più altezze per una singola coppia di coordinate orizzontali.

La Figura 1.2 mostra un esempio di mappa altimetrica, mentre la Figura 1.3 mostra come a partire da tale height map un motore di sviluppo possa produrre terreno tridimensionale.

Il progetto in esame fa utilizzo di una struttura dati che funge da mappa altimetrica.

1.2.6 Noise

In elettronica, il rumore (in inglese *noise*), anche chiamato interferenza o disturbo, è una forma di segnale elettrico indesiderato che può sommarsi al segnale originario, provocando perdite o alterazioni dell'informazione. Prendendo ispirazione da questo campo, si possono definire alcune funzioni matematiche che rappresentino questo rumore o disturbo. L'obiettivo nell'ambito della generazione procedurale è impiegare tali funzioni per la scelta dei punti in cui disporre gli elementi di gioco, in quanto con il loro andamento

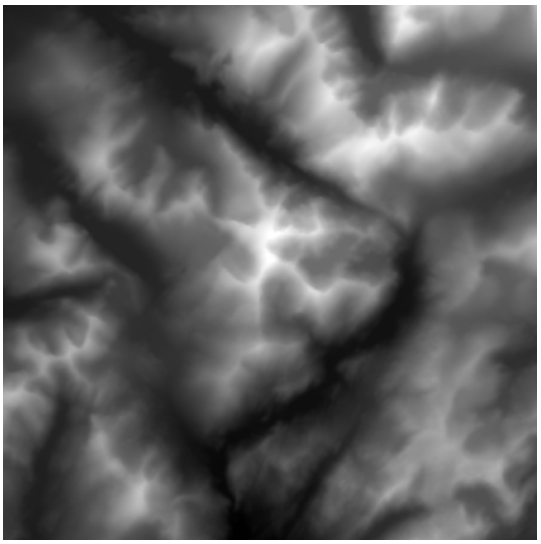

Figura 1.2: Rappresentazione in scala di grigi di una mappa altimetrica usata nei videogiochi. Il colore bianco rappresenta i punti più alti, quello nero i punti più bassi. Fonte: [6].

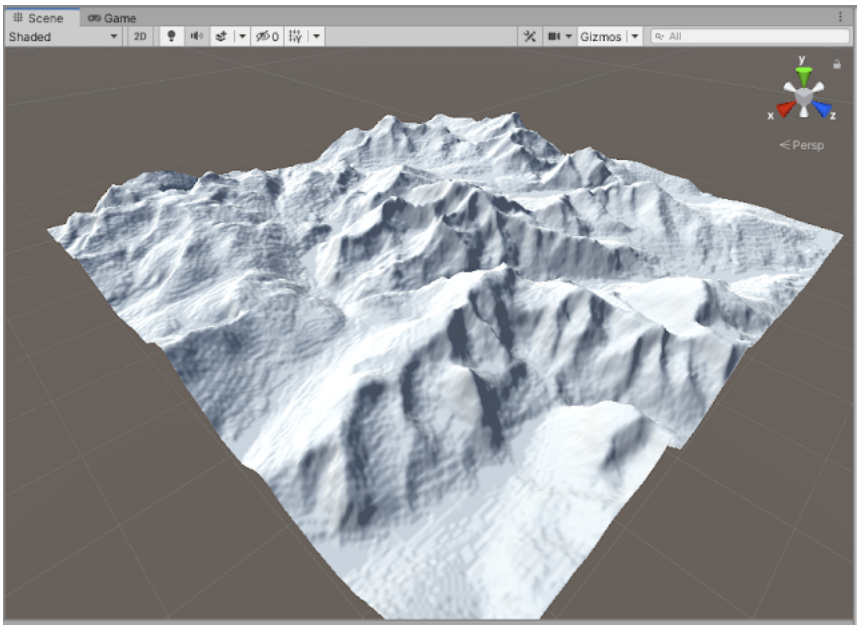

Figura 1.3: Mappa 3D ottenuta come risultato a partire dalla mappa in Figura 1.2. Fonte: [6].

irregolare sono idonee per rappresentare modelli che all'osservatore appaiano casuali e ricordino gli ambienti naturali.

Uno degli esempi di rumore forse più "visivamente" noti è il rumore statico presentato dalle TV analogiche, dovuto ai segnali casuali ricevuti dall'antenna priva di segnale o generati dall'elettronica del dispositivo. Questo causava un tremolio di punti bianchi e neri, che ricorda le immagini sfruttabili come base per una mappa altimetrica; tuttavia, questo tipo di trama è troppo irregolare nella maggior parte dei casi per poterne ricavare una mappa 3D adatta ai giochi.

Spesso più funzioni di rumore si possono sommare per ottenere comportamenti più complicati. Una tecnica adottata anche nel progetto e in generale quando si sfrutta il Perlin noise è combinare più *noise octave*. Le *noise octave* (ottave) sono funzioni che raddoppiano la frequenza e dimezzano l'ampiezza di una funzione di partenza base o di un'altra ottava. Prendono il nome dall'ambito musicale, in quanto una nota più alta di un'ottava rispetto ad un'altra è emessa al doppio della frequenza di quest'ultima.

Dato che tutte le ottave vengono sommate al rumore base di partenza, il fatto di diminuirne sempre più l'ampiezza (e dunque l'influenza sul risultato iniziale) consente di mantenere eventuali strutture abbozzate in precedenza, aggiungendo via via dettagli più fini. In termini pratici, se il rumore base è adatto per descrivere montagne, le ottave applicate potrebbero rappresentare alcuni avvallamenti o protuberanze rocciose, seguite dagli alberi, seguite dai detriti più fini. Il sito indicato nella fonte della Figura 1.4 può essere utile per dare un'idea più intuitiva del funzionamento di questi livelli sovrapposti di funzioni di interferenza.

1.2.7 Perlin noise

Introdotto per la prima volta nel 1984 da Ken Perlin, che lo aveva sviluppato con l'obiettivo di impiegarlo nella realizzazione di texture naturalistiche per il film *Tron* [13], l'algoritmo per il rumore di Perlin è vastamente adottato sia per la sua efficienza che per la qualità dei panorami che crea. Lo stesso ideatore vi ha iterato più volte, proponendo un'implementazione migliorata nel 2002 e un'alternativa chiamata *Simplex noise* successivamente.

L'algoritmo di Perlin può essere applicato in una, due, tre o quattro dimensioni, e richiede in input altrettante coordinate in virgola mobile, le quali **identificano univocamente un punto P** all'interno dello spazio. A partire da quel punto si trovano le coordinate intere del segmento, quadrato, cubo o ipercubo che lo contiene. **Per ciascuno dei vertici di questa figura** nella versione iniziale del 1984 **si creavano dei vettori gradiente** mediante generatori pseudo-casuali di numeri. La versione del 2002 rimuove questa necessità, e nell'esempio in tre dimensioni sfrutta i vettori (x, y, z) elencati in un semplice array. Questi vettori giacciono sulle rette che collegano il centro del cubo a ciascuno dei suoi 12 spigoli (nel caso 3D). Per ragioni di accessi ottimizzati alla

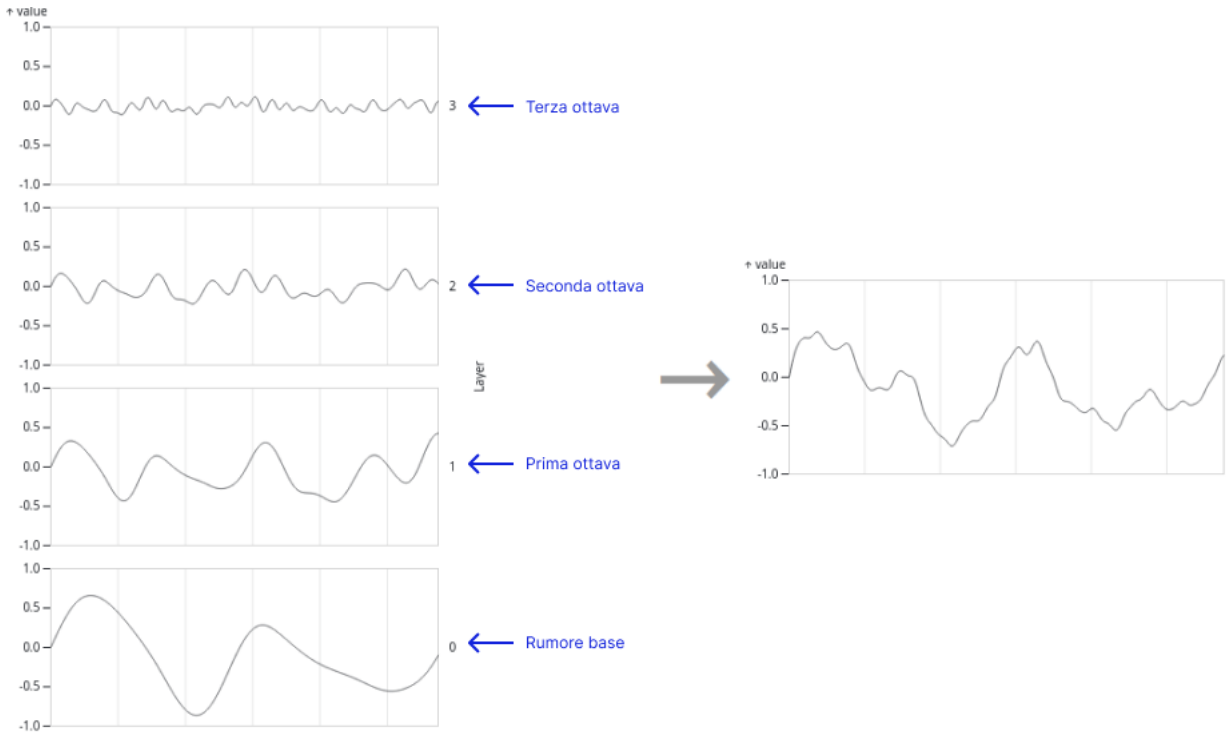

Figura 1.4: Combinazione di funzioni di *octave noise* per ottenere una funzione risultante più complessa. Fonte: [3].

memoria, il numero dei vettori gradiente memorizzati nell'array potrebbe essere portato a 16, in una maniera che non influenza la "casualità" e "naturalezza" del risultato finale.

Una volta scelto un vettore gradiente per ciascun vertice del cubo, si considera il **vettore distanza** che collega il punto di partenza P ai vertici del cubo unitario che lo contiene. Si effettua il prodotto scalare tra il gradiente scelto per quel vertice e quest'ultimo vettore così individuato: questa operazione produce come risultato un valore che possiamo chiamare *influenza* del gradiente sul vettore distanza.

Gli otto valori di influenza così ottenuti appartengono all'intervallo $[-1, 1]$, a seconda che il vettore distanza e quello gradiente puntino nella stessa direzione, in direzioni opposte, oppure che siano perpendicolari, caso in cui si ottiene 0 come valore di influenza. Si effettua una media pesata di tutti questi valori, nella quale il peso viene definito mediante interpolazione con un polinomio di 5° grado specificato da Ken Perlin:

$$6t^5 - 15t^4 + 10t^3.$$

Per *interpolazione* si intende considerare i valori noti di una funzione non nota all'interno di un intervallo e sfruttare un polinomio che approssima i punti noti, chiamato *polinomio interpolante*, per calcolare i valori non noti di tale funzione in tale intervallo. Il polinomio interpolante indicato da Perlin ha il vantaggio di non causare discontinuità e ammassamenti visibili quando i gradienti si trovano quasi allineati lungo gli assi. Per le implicazioni matematiche precise si rimanda all'articolo pubblicato dall'autore [17]. Non solo, ma la scelta di un polinomio di grado superiore al primo consente di ottenere paesaggi che "sfumano" in maniera credibile, e che non risultano frammentati come nel caso di un'interpolazione lineare.

Una guida completa all'implementazione dell'algoritmo di Perlin, corredata di immagini significative, si può consultare alla seguente fonte: [2].

Nel caso del codice da ottimizzare nell'ambito di questa tesi, l'autore ha fatto pesante uso del Perlin noise, combinando anche altre tecniche come ad esempio le ottave.

Capitolo 2

Tecnologie utilizzate

Verrà ora fornita una panoramica sugli strumenti utilizzati per la realizzazione della componente progettuale di questa tesi. Il progetto originale, scritto in C per la parte di dominio e in OpenGL per la parte di rendering, è stato concepito per eseguire in ambiente Unix. Vengono sfruttate diverse librerie, fra cui una per l'algebra lineare, una per la cattura degli input all'interno di una finestra e una per le funzioni di rumore. Per la compilazione si utilizza un Makefile.

Per quanto riguarda la parte di ottimizzazione, è stata adottata una scheda grafica NVIDIA che supporta CUDA (*Compute Unified Device Architecture*). Lo sviluppo è stato effettuato usando come IDE (*Integrated Development Environment*) CLion di JetBrains, e sfruttando gli strumenti da riga di comando del Cuda Toolkit (debugger e compute-sanitizer), di cui si parlerà diffusamente nelle sezioni di questo capitolo.

2.1 Cenni di programmazione parallela in CUDA

CUDA [14] è il nome dato ad un'architettura hardware e a un paradigma di programmazione dall'azienda NVIDIA. Con l'introduzione di CUDA si è segnato un importante passaggio dall'utilizzo delle schede grafiche (GPU) unicamente per scopi di elaborazione di immagini e video alla possibilità di impiegarne l'hardware per compiti di qualsiasi tipo (GPGPU, *General Purpose Graphics Processing Unit*).

Per comprendere i vantaggi offerti da una GPGPU, d'ora in poi abbreviato in GPU, è necessario prima capire che cosa sono i *thread* e cosa si intende per esecuzione concorrente e parallela di un programma. I thread sono sequenze di istruzioni. I processi, ossia i programmi attivi caricati nella memoria di un calcolatore, possono lanciare più di un thread per svolgere il proprio lavoro. Le istruzioni vengono solitamente eseguite da un'unità centrale, la CPU, che alterna i vari thread attivi eseguendo un po' per volta alcune istruzioni di ciascuna sequenza. Quando, come nel caso appena descritto, più thread vengono eseguiti da un'unica unità centrale (o processore, o *core*), si parla di

esecuzione concorrente. Sebbene si abbia l'impressione che i vari thread siano tutti eseguiti contemporaneamente, a basso livello il processore alterna molto rapidamente i thread attualmente in esecuzione, portando avanti i flussi di controllo di ciascuno un po' per volta.

A differenza dell'esecuzione concorrente, nell'**esecuzione parallela** invece si hanno più processori o unità di controllo, che possono eseguire a tutti gli effetti istruzioni contemporaneamente.

Le CPU, pensate per eseguire programmi dai flussi di controllo complessi e lavorare con diverse gerarchie di memoria improntate ad un caching molto articolato, sono adatte ad eseguire rapidamente decine di thread. A differenza delle CPU, le GPU sono improntate all'esecuzione efficiente di enormi quantità di calcoli semplici (cioè privi di numerosi salti condizionali o accessi complicati a diverse zone di memoria) mediante migliaia di thread eseguiti in parallelo su altrettanti piccoli processori o core. Il vantaggio di programmare per le GPU si ottiene quando si aumenta il *throughput*, ossia la quantità di calcoli che deve essere effettuata. Pensando al caso di un'immagine, una singola CPU sarebbe costretta a calcolare sequenzialmente, pixel per pixel, l'intera immagine prima di poterla mostrare a schermo. In una GPU, la numerosità dei thread consente a ciascuno di essi di calcolare un solo pixel in una data posizione dell'immagine, e tutto questo avviene in contemporanea.

In linea generale, quando ci si trova a dover effettuare una grande quantità di operazioni semplici in maniera *indipendente* (ossia quando un thread A può lavorare senza dipendere dal risultato prodotto da un thread B, cosa che richiede meccanismi di sincronizzazione che inevitabilmente rallentano l'esecuzione) è possibile prendere seriamente in considerazione l'ottimizzazione tramite calcolo parallelo.

Per quanto riguarda il paradigma di programmazione, CUDA è un'estensione del linguaggio C/C++, che ne supporta la maggior parte dei meccanismi. Una volta individuata un'opportunità di parallelizzazione nel codice *seriale* (cioè pensato per una singola unità di esecuzione), la struttura tipica dei programmi CUDA è la seguente:

- viene copiata una certa quantità di dati dalla memoria riservata alla CPU, chiamata *host*, alla memoria riservata alla GPU, chiamata *device*;
- il device esegue dei calcoli paralleli sui dati ricevuti;
- l'host si sincronizza con il device, cioè attende che il device abbia finito il proprio lavoro;
- si copia il risultato prodotto dal device dalla sua memoria riservata a quella dell'host.

Per un'ottimizzazione effettiva, è bene ridurre al minimo i punti di sincronizzazione e di copiatura di dati fra host e device.

2.2 Introduzione di codice CUDA in un progetto in C e compilazione tramite Makefile

In linea con le buone norme di programmazione, per assicurare la modularità del codice, ossia la possibilità di cambiare facilmente i componenti del progetto, il codice CUDA può essere nascosto dietro a delle API definite in file header. L'idea è estendere il progetto originale con una libreria di funzioni i cui prototipi sono dichiarati in un file .h e implementare tali funzioni in dei file .cu adibiti a contenere il codice CUDA.

Questa soluzione ha il vantaggio di consentire ai moduli iniziali di essere compilati mediante le istruzioni già preparate dall'autore (il cui username GitHub è **jdah**), sfruttando un compilatore apposito per il linguaggio C, in questo caso **clang**. I file .cu invece possono essere compilati dal compilatore CUDA ufficiale, **nvcc**, e sempre ad esso si può delegare la fase di linking.

Affinché il codice C possa effettuare delle chiamate a codice CUDA, è importante definire in tutti i file CUDA il simbolo `extern "C"` e aprire un blocco di graffe, come mostrato nel Listato 2.1.

```
#ifdef __cplusplus
extern "C" {
#endif

    // Qui viene inserito il codice CUDA

#ifdef __cplusplus
}
#endif
```

Listato 2.1: Inserimento di codice CUDA in un progetto in C.

Fatto ciò, si può scegliere un normale file .h incluso nel progetto C e scrivere tutti i prototipi delle funzioni che verranno poi definite nei file .cu. Una caratteristica peculiare di CUDA è il fatto di separare le funzioni definite per host e quelle per device. È possibile separare la dichiarazione dei prototipi di funzioni per device e la loro implementazione effettiva su più file, ricordandosi di specificare l'opzione **-dc** quando si compila con nvcc. -dc infatti sta per *device code* e consente di delegare al linker **nvlink**, usato automaticamente dal comando nvcc, il compito di collegare i nomi delle funzioni alla loro implementazione.

Segue una struttura in pseudo-codice di un possibile Makefile per l'automazione della compilazione di un progetto di questo tipo, visibile nel Listato 2.2. Questo esempio è una semplificazione del Makefile in uso nel progetto di jdah, ed è pensato per la compilazione in modalità **debug**. Tale modalità consente di incorporare nell'eseguibile i simboli per

il debugging (come ad esempio i numeri di riga e i sorgenti in cui sono presenti le varie funzioni), e in genere disabilita tutte le ottimizzazioni.

```
C_COMPILER = clang
CUDA_COMPILER = nvcc

# L'opzione -g abilita i simboli di debug
C_FLAGS = -std=c11 -O3 -g -Wall -Wextra -Wpedantic

# L'opzione -G abilita i simboli di debug per il codice
   del device
CUDA_FLAGS = -g -G
LDFLAGS = [...] -lm # flag per il linker C

SRC = [...] # nomi di tutti i sorgenti C

# Contiene la lista di tutti i file oggetto da creare,
# i cui elementi sono ottenuti sostituendo
# <<.o>> a <<.c>> negli elementi della variabile SRC
OBJ = $(SRC:.c=.o)

# Come prima, ma per i file .cu
CUDA_SRC = [...]
CUDA_OBJ = $(CUDA_SRC:.cu=.o)

OBJ += $(CUDA_OBJ) # unione di tutti i file oggetto

# Target del Makefile
all: $(OBJ) gpuCode.o
    $(CUDA_COMPILER) -o main.exe $^ $(LDFLAGS)

# Regola per creare tutti i file oggetto a partire dai
   file .c
%.o: %.c
    $(C_COMPILER) -o $@ -c $< $(CFLAGS)

# Regola per creare tutti i file oggetto a partire dai
   file .cu
# -x cu specifica che il linguaggio da usare per la
   compilazione e' CUDA; e' un flag opzionale
# -dc specifica che i simboli come i nomi di funzione
```

```
    del codice per device dovranno essere risolti dal
    linker in una fase successiva
%.o: %.cu
    $(CUDA_COMPILER) -x cu -dc $< -o $@ $(CUDA_FLAGS)

# Regola per effettuare un linking preliminare del solo
    codice CUDA
# -dlink risolve le dipendenze fra nomi nel codice per
    device (i cosiddetti kernel)
# -arch=sm_xx specifica l'architettura della GPU NVIDIA
    da usare per il linking; varia da GPU a GPU
gpuCode.o: $(CUDA_OBJ)
    $(CUDA_COMPILER) -arch=sm_75 -dlink $(CUDA_OBJ) -o
       gpuCode.o -L/opt/cuda/nvvm/lib64 -lcudart

.PHONY: clean

clean:
    -rm -rf $(OBJ) *.exe
```

Listato 2.2: Esempio di Makefile per un progetto misto in linguaggio C e CUDA. Dove è presente la dicitura [...] si deve intendere che alcuni elementi sono stati omessi per semplicità.

Il Makefile si compone di una sezione in cui si definiscono i due diversi compilatori mediante variabili, e i vari flag da passare a ciascuno di essi. Segue una parte in cui si elencano tutti i file sorgente presenti nel codice e si crea una variabile `OBJ` che conterrà tutti i file oggetto da generare con la compilazione. Successivamente si effettua la compilazione separata dei file C mediante il `C_COMPILER` e dei file CUDA con il `CUDA_COMPILER`. I file oggetto generati dalla compilazione dei file CUDA vengono linkati separatamente in maniera preliminare, fase visibile nel Listato 2.2 in corrispondenza del target `gpuCode.o`. In questa fase può essere utile specificare il parametro opzionale `arch=sm_xx`, che indica l'architettura della propria scheda grafica NVIDIA. Per trovare la *compute capability*[1] da sostituire a `xx`, consultare il sito ufficiale: https://developer.nvidia.com/cuda-gpus. In seguito, scrivere al posto di `xx` la compute capability senza il punto.

Nel Makefile sono presenti alcuni simboli preceduti da `$`:

- `$@`: nome del target del Makefile;
- `$<`: primo prerequisito per generare questo target, solitamente un file sorgente;

[1] La *compute capability* può essere considerata un numero di versione della scheda grafica, che identifica le funzionalità di cui dispone. Per maggiori dettagli, consultare il sito ufficiale di NVIDIA: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#compute-capability.

- `$^`: nomi di tutti i prerequisiti del target, separati da uno spazio [1].

Nota: potrebbe tornare comodo verificare i simboli inseriti nei file oggetto ed eseguibili. Per fare ciò si sfrutta il comando `nm nome-file`.

2.3 Guida pratica all'uso di *cuda-gdb*

Non sempre è possibile impiegare un IDE per lo sviluppo e soprattutto il debugging delle applicazioni CUDA. Ad esempio, CLion non offre soluzioni soddisfacenti per gli sviluppatori Windows e MacOS, mentre per Linux i passi proposti nella guida non sembrano funzionare [11]. Per questo, saper usare il debugger ufficiale di CUDA, **cuda-gdb**, può essere di importanza essenziale.

cuda-gdb è uno strumento da riga di comando, installato automaticamente assieme al CUDA Toolkit, che prende come parametro in input un qualsiasi eseguibile e consente di inserire breakpoint anche nelle funzioni eseguite sul device, siano queste *kernel* (cioè funzioni per device chiamabili dall'host) o normali funzioni (chiamabili solo da altri kernel o altre funzioni esclusivamente per device). cuda-gdb consente anche di visualizzare le variabili nella memoria del device e di spostarsi da un CUDA thread ad un altro secondo le necessità.

In questa sezione è presente una breve raccolta di comandi utili, che in fase di sviluppo del progetto si sono rivelati sufficienti per le mie esigenze.

2.3.1 Comandi base

- `cuda-gdb ./nome-eseguibile`: avvia una sessione di debug per l'eseguibile specificato; **i comandi successivi sono da lanciare all'interno della shell di comandi aperta da questo comando iniziale**;

- `help`: mostra i capitoli in cui è suddiviso il manuale del debugger; fare `help nome-capitolo` consente poi di vedere la lista di comandi per quello specifico insieme di azioni e le loro spiegazioni, mentre `help nome-comando` spiega più nei dettagli l'utilizzo del singolo comando;

- `break nome-funzione`/`break nome-sorgente:numero-riga`: inserisce un breakpoint all'inizio della funzione specificata da `nome-funzione`, oppure alla riga del sorgente indicato. Esempi di comandi sono `break my_kernel` e `break my_file:112`;

- `run`: esegue il programma dall'inizio fino al primo breakpoint;

- `continue`: riprende l'esecuzione da dove si era interrotta e si arresta al prossimo breakpoint;

- `step`: esegue la riga di codice corrente e si arresta prima di eseguire quella successiva;
- `exit`: termina l'esecuzione del programma e la sessione di debug, uscendo da cuda-gdb.

2.3.2 Mostrare variabili ed espressioni

- `display expression`: consente di stampare a video il valore di `expression` ogni volta che l'esecuzione si arresta; `expression` può essere una qualunque espressione, dal nome di una variabile a un'operazione matematica;
- `print expression`: simile a `display`, consente di stampare il valore di un'espressione, ma lo fa una sola volta;
- `print/display *(nome-array)@N`: mostra i primi `N` valori dell'array `nome-array`;
- `watch expression`: imposta un watchpoint, e arresta l'esecuzione del programma ogni volta che il valore di `expression` cambia;
- `call nome-funzione(argomenti)`: mostra il risultato della chiamata alla funzione indicata, anche se personalmente ho avuto problemi a far funzionare questo comando, in quanto spesso non trovava la funzione a cui mi riferivo;

2.3.3 Stack di chiamate a funzione

- `where`: restituisce lo stacktrace attuale. Ogni elemento all'interno dello stacktrace è chiamato *frame* dello stack;
- `frame apply N comando`: applica il comando `comando` su `N` frame dello stack, a partire da quello corrente. Può essere utile per vedere il valore di variabili prima e dopo la chiamata ad una funzione;
- `frame apply level N comando`: applica il comando `comando` soltanto al frame `N` dello stacktrace corrente.

2.3.4 Mostrare e navigare fra i CUDA thread

- `info cuda blocks`: stampa il numero di blocchi di thread attivi al momento;
- `info cuda threads`: stampa il numero di thread attivi al momento, suddivisi per blocco, e indica a quale riga di codice è ferma l'esecuzione di ciascun thread;
- `cuda thread (x, y, z)`: si sposta l'attenzione sul thread specificato alle coordinate (x, y, z), deducibili mediante i comandi precedenti.

2.4 Guida pratica all'uso di *compute-sanitizer*

Il CUDA Toolkit mette a disposizione un altro utile strumento per lo sviluppo, chiamato **compute-sanitizer**. Si tratta di un tool capace di rilevare accessi a memoria riservata, utilizzo di memoria globale non inizializzata, race condition e errori di sincronizzazione fra i CUDA thread. Similmente a cuda-gdb, anche il comando compute-sanitizer prende in input un eseguibile, su cui poi possono essere effettuate diverse operazioni di controllo.

Il compute-sanitizer viene installato nella cartella extras del percorso dove viene messo il CUDA Toolkit. Sul mio sistema, il percorso completo è `/opt/cuda/extras/compute-sanitizer`.

2.4.1 Comandi

- `compute-sanitizer --help`: mostra la lista dei comandi disponibili e il loro utilizzo.
- `compute-sanitizer ./nome-eseguibile`: avvia l'eseguibile e controlla che non ci siano accessi a memoria riservata durante l'esecuzione (questo è il comportamento di base);
- `compute-sanitizer --tool nome-tool ./nome-eseguibile`: consente di specificare uno fra quattro tool disponibili al posto di `nome-tool`. I tool disponibili sono:
  - `memcheck`: tool di base, controlla accessi alla memoria;
  - `racecheck`: controlla se ci siano race condition nella memoria condivisa dei thread di uno stesso blocco;
  - `synccheck`: verifica eventuali problemi di sincronizzazione fra thread di uno stesso blocco;
  - `initcheck`: controlla che tutta la memoria globale utilizzata venga prima inizializzata.
- `compute-sanitizer --print-limit 0 ./eseguibile`: non pone alcun limite al numero di stampe che possono essere effettuate dal compute-sanitizer. Al posto di 0 è possibile specificare un qualsiasi altro numero intero positivo.

Capitolo 3

Struttura del codice da ottimizzare

In questo capitolo verrà svolta un'analisi del codice seriale di partenza, presentando gli aspetti salienti del progetto e delineando il processo che ha portato all'individuazione di un'opportunità di intervento per la parallelizzazione.

Per seguire adeguatamente i riferimenti in questa parte, potrebbe essere utile fare riferimento al codice originale: https://github.com/jdah/minecraft-weekend. L'autore ha anche creato un video su YouTube in cui ha mostrato la realizzazione del proprio clone di Minecraft in due giorni: https://www.youtube.com/watch?v=4OO_-1NaWnY.

3.1 Struttura delle cartelle

È presente un Makefile generale nella cartella radice del progetto, mentre i file sorgente sono tutti nella cartella src. La cartella lib contiene cinque librerie statiche con diversi scopi (calcoli algebrici, generazione del rumore, rendering). Ciascuna di queste librerie è compilata mediante un Makefile più interno, chiamato da quello principale.

Nella cartella src, si nota un file principale chiamato main.c e un'ulteriore suddivisione in cartelle (block, entity, gfx, ui, util e world). La cartella world contiene alcuni file generali e una sottocartella gen, che contiene i sorgenti per la logica di generazione procedurale del mondo, i quali saranno di grande interesse per lo scopo di questa tesi.

3.2 Game loop

Nel file principale è presente una funzione `main()` che si occupa di inizializzare i vari componenti dell'applicazione e poi di dare il via ad un ciclo infinito, chiamato *game loop*. Il game loop è un pattern di programmazione adottato spesso nei videogiochi in tempo reale, in cui ci sono fenomeni che devono accadere indipendentemente dagli input immessi dal giocatore.

Il funzionamento di base è il seguente: si decide un numero di frame da mostrare a schermo ad ogni secondo, chiamato *framerate* o *frames per second (FPS)*. Ciascun frame è un fotogramma del mondo di gioco, e nei giochi in tempo reale l'intero mondo visibile è ricalcolato una volta per ogni frame. Tipicamente, il valore è 60 FPS, il che significa che ciascun frame deve essere calcolato entro $16.\overline{6}$ ms. Deciso il framerate, all'interno del ciclo principale si effettuano le seguenti operazioni:

- si controllano e processano eventuali input del giocatore;
- si manda avanti la logica del mondo, la quale potrebbe includere calcoli di natura fisica, scritture su file, generazione procedurale, emissione di suoni o gestione dell'intelligenza artificiale delle entità, ecc.;
- si elabora la scena da mostrare a schermo. Se i calcoli da effettuare sono troppo onerosi per rispettare il framerate, questa fase viene in genere lasciata indietro.

Un esempio di game loop è visibile nel Listato 3.1. Per assicurare il corretto funzionamento della logica del gioco, si controlla dopo ogni ciclo del game loop di quanto l'orologio "del gioco" è più indietro rispetto al tempo reale. Se questa quantità, chiamata *lag*, è maggiore del tempo corrispondente ad un frame, allora bisogna rimandare la fase di rendering e mandare avanti la logica del gioco fino alla sincronizzazione con il tempo reale. Se invece non c'è nessun ritardo, si salta la fase di aggiornamento di logica del gioco e si passa direttamente al rendering, cosa che consente potenzialmente di disegnare più frame rispetto a quelli previsti. Questa soluzione consente a macchine di potenza e velocità diverse di offrire un'esperienza di gioco simile.

In sintesi, il game loop è un costrutto che consente di acquisire gli input del giocatore in maniera non bloccante (in quanto il mondo di gioco va avanti a prescindere) e di aggiornare la logica di gioco con un framerate costante indipendente dall'hardware su cui il gioco è in esecuzione [15].

```
double previous = getCurrentTime();
double lag = 0.0;
while (true) {
    double current = getCurrentTime();
    double elapsed = current - previous;
    previous = current;
    lag += elapsed;

    processInput();

    /* Finche' il lag, cioe' il tempo trascorso tra un
       ciclo e un altro del game loop, e' maggiore del
       tempo previsto per calcolare un frame, allora
       manda avanti solo la logica del gioco */
```

```
        while (lag >= MS_PER_UPDATE) {
            update(); /* metodo che aggiorna il mondo di
               gioco, mandandone avanti la logica */
            lag -= MS_PER_UPDATE;
        }

        render();
    }
```

Listato 3.1: Esempio di game loop. Da notare il fatto che la funzione `render()` viene chiamata più volte nel caso in cui avanzi del tempo prima di dover calcolare il prossimo frame. Fonte: [15]

3.3 Entità del dominio e funzionamento generale

Le entità principali del dominio sono il **giocatore** e il **mondo**; quest'ultimo è a sua volta suddiviso in **chunk**, che sono gruppi di **blocchi**, cioè le singole unità a forma di cubo che compongono tutto l'ambiente di gioco. Infine esistono le **decorazioni**, un modo sintetico di chiamare alberi, fiori e altri piccoli elementi che possono essere presenti sulla superficie del mondo. jdah ha modellato queste entità mediante delle struct che possono essere rinvenute nelle interfacce contenute nella cartella world (file world.h e chunk.h).

Ai fini della generazione procedurale non ci interessa come viene gestito il giocatore e ci focalizziamo sul mondo.

Il mondo è composto da un seme unico, utilizzato nella generazione procedurale. Infatti a partire da uno stesso seme si otterrà sempre lo stesso mondo. Segue un array di height map, di cui abbiamo discusso nel Capitolo 1, e un array di chunk. Ciascuna heightmap è composta da 32×32 elementi, che rappresentano le colonne di blocchi all'interno di un chunk, e ciascun chunk è composto da $32 \times 32 \times 32$ blocchi. Il numero 32 è una costante definita globalmente, chiamata `CHUNK_SIZE` (chunk.h). In un'altra costante presente nella struct World si imposta il numero di chunk che devono essere generati contemporaneamente ad un limite fisso di 16 (world.c). Questi numeri potrebbero essere aumentati per studiare il comportamento dell'applicazione all'aumentare del lavoro da svolgere.

Ad ogni frame, la funzione `window_loop()` si occupa di raccogliere e processare gli input del giocatore, di aggiornare il mondo di gioco e di renderizzarlo, in linea con quanto ci si aspetta dalla struttura di un game loop. Il giocatore può interagire con il mondo muovendosi, volando, distruggendo i blocchi presenti o piazzandone altri. Il game loop è complicato da un'ulteriore possibilità di mandare avanti il tempo di gioco velocemente, cosa che consente di far calare la notte.

A partire dalla posizione del giocatore, si itera su tutti i chunk presenti nel mondo e si calcolano eventuali chunk vuoti. Si considera la posizione del giocatore e si calcolano i 16 chunk da mostrare; se il giocatore si sposta al di fuori del chunk corrente, quelli più indietro vengono rimossi dal mondo e ne vengono aggiunti altri nella direzione in cui si sta muovendo. In questo modo, il mondo visualizzato mantiene sempre il giocatore al centro.

Il calcolo dei chunk avviene tramite un algoritmo di generazione procedurale che sfrutta una mappa altimetrica per ciascun chunk e combina tecniche di Perlin noise e octave noise (Capitolo 1) per generare paesaggi dagli aspetti credibili. Data la possibilità di calcolare dinamicamente ciascun chunk, non c'è bisogno di mantenere in memoria o su file i chunk già generati. Il gioco può mostrare una varietà di biomi, fra cui spiagge, oceani, foreste, savane e montagne. Non sono presenti caverne, com'era prevedibile, dato l'impiego delle mappe altimetriche. I biomi sono distinguibili grazie alla presenza di diversi tipi di blocco (terra, pietra, acqua, legno, neve). La generazione procedurale fa sì che non si noti la transizione da un chunk ad un altro, contribuendo alla sensazione di immersione mentre si esplora il mondo.

L'implementazione attuale non consente di memorizzare gli interventi del giocatore sui chunk nel caso in cui questi vengano lasciati; per fare ciò sarebbe necessario tenere traccia delle modifiche di un giocatore in qualche modo. Il gioco Minecraft lo fa mediante salvataggio di file su disco. Inoltre, è presente un bug che fa crashare il programma quando il giocatore scava troppo in profondità. La risoluzione di questi problemi esula dallo scopo della tesi.

3.4 Individuazione del codice da ottimizzare

Considerando la struttura del dominio, è naturale aspettarsi che in un qualche punto del codice i vari chunk vengano popolati con dei blocchi in qualche maniera. Seguendo la catena di chiamate a funzione a partire dalla funzione di inizializzazione, si arriva alla funzione `world_load_chunk()` definita in world.c. Questa funzione carica un nuovo chunk in memoria, chiamando `worldgen_generate()` per generare i blocchi al suo interno, e poi controlla se sono presenti blocchi in attesa per questo chunk. I blocchi in attesa sono quei blocchi che vengono creati al momento della creazione di una decorazione, ma che fanno parte di un chunk che non è stato ancora caricato. Questi blocchi vengono memorizzati in una struct `unloaded_blocks` all'interno della struct World.

La seguente tabella riassume le chiamate a funzione per arrivare a `worldgen_generate()`, che si occupa proprio della parte di generazione procedurale di nostro interesse. Le chiamate a funzione sono ordinate cronologicamente:

Nome funzione	File
`init()`	main.c
`world_init()`	world/world.c
`world_set_center()`	world/world.c
`load_empty_chunks()`	world/world.c
`world_load_chunk()`	world/world.c
`worldgen_generate()`	world/gen/worldgen.c

Il Listato 3.2 mostra uno pseudo-codice dell'implementazione originale della funzione `worldgen_generate()`.

```
1   void worldgen_generate(struct Chunk *chunk) {
2       /* Inizializzazione di un generatore casuale di numeri usato
           per la creazione delle decorazioni */
3       [...]
4
5       struct Heightmap *heightmap = chunk_get_heightmap(chunk);
6       if (!heightmap->flags.generated) {
7           heightmap->flags.generated = true;
8
9           // Creazione delle struct noise
10          [...]
11
12          // Per tutte le colonne della height map:
13          for (s64 x = 0; x < CHUNK_SIZE.x; x++) {
14              for (s64 z = 0; z < CHUNK_SIZE.z; z++) {
15                  s64 world_x = chunk->position.x + x, world_z =
                        chunk->position.z + z;
16
17                  /* Si calcolano valori per l'altezza (h), l'
                        umidita' (m), la temperatura (t), la ruvidita
                        ' (r) e la perturbazione specifica dei biomi
                        di montagna (n e p) sfruttando le struct
                        noise istanziate prima */
18                  f32 h, m, t, r, n, p = [...]
19
20                  // Si usano i valori trovati per determinare il
                        bioma presente in questa colonna del chunk
21                  enum Biome biome_id = get_biome(h, m, t, n, n +
                        h);
22                  struct BiomeData biome = BIOME_DATA[biome_id];
23
24                  /* Si impostano i valori dei WorldgenData all'
                        interno della heightmap. h_b rappresenta un'
```

```
                    altezza data dal bioma, b e' l'id del bioma.
                    */
25                  heightmap->worldgen_data[x * CHUNK_SIZE.x + z] =
                        (struct WorldgenData) {
26                      .h_b = [...],
27                      .b = biome_id
28                  };
29              }
30          }
31
32          // si smussa l'altezza di ciascuna colonna di blocchi
               facendo una media aritmetica con le altezze delle
               colonne vicine
33          for (s64 x = 0; x < CHUNK_SIZE.x; x++) {
34              for (s64 z = 0; z < CHUNK_SIZE.z; z++) {
35                  f32 v = 0.0f;
36                  v += (get_data_at(x - 1, z - 1)).h_b;
37                  v += (get_data_at(x + 1, z - 1)).h_b;
38                  v += (get_data_at(x - 1, z + 1)).h_b;
39                  v += (get_data_at(x + 1, z + 1)).h_b;
40                  v *= 0.25f;
41                  get_data_at(x, z).h = v;
42              }
43          }
44      }
45
46      for (s64 x = 0; x < CHUNK_SIZE.x; x++) {
47          for (s64 z = 0; z < CHUNK_SIZE.z; z++) {
48              // Per ogni colonna nella heightmap, si prende l'
                   altezza h della colonna e l'id del bioma della
                   colonna
49              struct WorldgenData data = heightmap->worldgen_data[
                   x * CHUNK_SIZE.x + z];
50              s64 h = data.h;
51              enum Biome biome = data.b;
52              struct BiomeData biome_data = BIOME_DATA[biome];
53
54              [...]
55
56              // Per ogni blocco in questa colonna (x, z):
57              for (s64 y = 0; y < CHUNK_SIZE.y; y++) {
58                  s64 world_y = chunk->position.y + y;
59
```

```
60                      enum BlockId block = [...] // si decide qual e'
                           il blocco adatto; se si tratta di un blocco d
                           'aria, si salta questo passaggio e si va all'
                           iterazione successiva
61
62                      // si imposta il blocco nella posizione corrente
63                      chunk_set_block(chunk, (ivec3s) {{ x, y, z }},
                           block);
64
65                      // Se l'altezza corrente coincide con la
                           superficie del terreno, genera delle
                           decorazioni con una certa probabilita'
66                      if (world_y == h) {
67                          // Codice per le decorazioni
68                          [...]
69                      }
70                  }
71              }
72          }
73      }
```

Listato 3.2: Pseudo-codice della funzione `worldgen_generate()`, responsabile della generazione procedurale dei blocchi all'interno di un singolo chunk, così com'è stata pensata da jdah.

Come si può vedere anche dallo pseudo-codice, la funzione non è di immediata comprensibilità, ma è fondamentale capire che cosa fa per riuscire ad applicare le modifiche desiderate.

Dato un chunk in input, se ne ricava la mappa altimetrica associata. Si controlla se in questa mappa altimetrica sono già stati generati i dati che indicano qual è l'altezza di ciascuna colonna di blocchi data una coppia di coordinate (x, z). I dati di questo tipo sono chiamati `WorldgenData` e hanno la forma indicata nel Listato 3.3.

```
struct WorldgenData {
    f32 h_b; // altezza data dal bioma
    s64 h, b; // altezza finale e id del bioma
}
```

Listato 3.3: Struttura di un dato di tipo `WorldgenData`, contenuti nella height map.

Se i `WorldgenData` sono già stati creati per questo chunk, allora si salta alla fase successiva (riga 6). Altrimenti, bisogna crearli sfruttando le tecniche di Perlin noise discusse in precedenza. Con un doppio ciclo `for` annidato si itera su tutti gli elementi della mappa altimetrica e si calcolano una serie di valori che rappresentano caratteristiche

di uno specifico bioma, quali l'altezza, l'umidità, la temperatura, l'eventuale altezza delle montagne presenti, ecc. La funzione `get_biome()` prende in input questi valori e restituisce l'id del bioma corrispondente, da mettere all'interno del campo `.b` della struct `WorldgenData`. Sempre a partire dal bioma e dai dati numerici individuati, è possibile determinare anche un'altezza "temporanea" della colonna di blocchi, da memorizzare nel campo `.h_b`. Questa altezza non è quella definitiva perché altrimenti si rischiano dei distacchi molto evidenti rispetto alle colonne vicine; per questo si effettua una fase finale di smussatura, in cui si calcola l'altezza finale v da mettere nel campo `.h` come la media delle altezze delle quattro colonne vicine in diagonale a quella attualmente presa in considerazione (righe 33–43). La funzione `get_data_at()` garantisce di prendere valori contenuti in questo chunk, restituendo i `WorldgenData` di una colonna sul bordo nel caso i vicini richiesti siano fuori dai bordi del chunk.

Una volta creata o recuperata la height map con i dati generati, si itera su tutte le coppie di coordinate (x, z) al suo interno e si controlla l'altezza `.h` e il bioma di riferimento `.b` per tali coordinate. Successivamente, si itera dal basso verso l'alto per tutti i blocchi che formano la colonna, piazzando di volta in volta il blocco di tipo giusto (a meno che si tratti di un blocco d'aria, che non viene piazzato per efficienza). Infine, quando si piazza l'ultimo blocco sulla superficie della colonna, c'è una probabilità casuale che si generi una decorazione (albero, cespuglio, fiore, ecc.).

Come nota aggiuntiva, si vede che l'autore ha utilizzato tipi di dati numerici diversi da quelli che si incontrano di solito nei linguaggi (`s64`, `f32`, ecc.). Il file types.h dentro la cartella util contiene le definizioni di questi tipi numerici (Listato 3.4). Questa è in realtà una prassi abbastanza comune quando si vuole assicurare portabilità dei tipi di dati numerici in C, in quanto le normali keyword (`long, long long, float`, ecc.) non assicurano che si utilizzi lo stesso numero di bit fra piattaforme diverse. Dal momento che per la manipolazione dei `WorldgenData` l'autore originale effettua operazioni bitwise in cui è essenziale avere un numero di bit fissato, si rende necessario adottare queste strategie.

```
#ifndef TYPES_H
#define TYPES_H

// fixed width numeric types
#include <stdint.h>

typedef uint8_t u8;
typedef uint16_t u16;
typedef uint32_t u32;
typedef uint64_t u64;

typedef int8_t s8;
```

```
typedef int16_t s16;
typedef int32_t s32;
typedef int64_t s64;

typedef float f32;
typedef double f64;

#endif
```

Listato 3.4: Contenuto del file types.h. L'autore ha ridenominato alcuni tipi numerici di dimensione costante (in byte) fra tutte le piattaforme.

In sintesi, la funzione `worldgen_generate()`:

- prende in input un chunk;
- controlla se esiste una height map per quel chunk;
- se non esiste, crea delle strutture Noise usate per la generazione procedurale, dopodiché la crea;
- se esiste o è appena stata creata, usa la height map per generare tutti i blocchi del chunk sulla base dei biomi che vi sono presenti;
- crea delle decorazioni con una probabilità casuale sulla superficie del terreno generato.

3.5 Considerazioni

La funzione `worldgen_generate()` presenta diversi cicli for annidati, che saltano all'occhio quando si cercano opportunità di parallelizzazione. Infatti, riprendendo l'esempio del calcolo di un'immagine, se una CPU deve farlo pixel per pixel, una GPU può effettuare questo calcolo sfruttando il parallelismo di massa. Ad ogni thread si assegna un solo pixel, e tutti i pixel vengono calcolati in contemporanea, riducendo drasticamente il tempo richiesto per l'elaborazione di un'immagine. Il fatto che qui si stia parlando di blocchi all'interno di un chunk presenta sicuramente delle analogie con il caso delle immagini.

Il passaggio successivo per assicurarsi che esista un'opportunità effettiva di parallelizzazione consiste nel cercare eventuali dipendenze fra i calcoli dei singoli blocchi. In altre parole: un blocco a delle coordinate (x, y, z) fissate dipende in qualche modo da un altro blocco a delle coordinate (x', y', z')?

Queste dipendenze esistono nel caso delle decorazioni, il cui codice è stato omesso nell'esempio fornito per brevità. Ad ogni modo, nel lavoro svolto **non** si è cercato di

parallelizzare anche la generazione delle decorazioni, in quanto proprio la presenza di tali dipendenza tra blocchi, possibilmente anche distribuiti fra chunk diversi, avrebbe complicato di molto il problema da risolvere.

Una dipendenza che invece non è trascurabile è quella del **calcolo di v quando si genera la height map** (righe 36-39 del Listato 3.2). L'operazione di smussatura delle altezze delle colonne di blocchi viene effettuata considerando i valori delle colonne vicine all'interno dello stesso chunk. Se, come accadrà applicando il paradigma CUDA, l'altezza `h_b` di ciascuna colonna viene calcolata da un thread diverso, allora conoscere i valori dei vicini potrà essere fatto **soltanto in seguito ad una sincronizzazione dei thread** che lavorano su uno stesso chunk; in altre parole, prima tutti i thread dovranno completare il calcolo di tutti i valori `h_b`, e solo poi sarà possibile usare tali valori per il calcolo dell'altezza media v.

A parte questo importante dettaglio, ciascun thread è poi libero di calcolare il proprio blocco all'interno del chunk senza dipendenze da altri thread, una volta che ci saremo assicurati di aver creato la heightmap corretta (calcolo che in gergo tecnico rientra nella categoria dei problemi *embarrassingly parallel*). Questo fatto potrebbe consentire un bel miglioramento dal punto di vista dell'efficienza del programma parallelo rispetto a quello seriale.

Capitolo 4

Parallelizzazione del codice su GPU

Affronteremo ora il succo della tesi, ossia l'ideazione di codice parallelizzato per GPU che consenta di ottimizzare la generazione procedurale del mondo a blocchi.

I contenuti verranno riportati con un approccio cronologico, descrivendo i problemi incontrati man mano e le soluzioni applicate. Tali problemi non saranno limitati soltanto al codice parallelo in senso stretto, bensì si prenderanno in considerazione tutte le difficoltà salienti affrontate durante il lavoro con un progetto di questo calibro.

Può essere utile seguire questa parte facendo riferimento al mio codice presente a questo indirizzo: https://github.com/Martinetto33/minecraft-weekend.

4.1 Piano generale

Si vuole parallelizzare la funzione di generazione procedurale `worldgen_generate()`. Per farlo, bisogna occuparsi di questi passaggi:

- Istanziare delle struct noise sul device. Ho notato che jdah le ricrea con gli stessi parametri ogni volta che deve generare una nuova height map, e ho voluto evitare questo spreco inizializzandole una sola volta nella memoria globale del device;
- Generare la height map se necessario; l'obiettivo è fare che ogni CUDA thread generi la propria istanza di `WorldgenData` per la propria posizione (x, z). Per farlo avrà bisogno di accedere in lettura alle struct noise, ma questo avviene senza problemi di race condition. La generazione della height map deve tenere conto della necessità di sincronizzazione fra i thread prima del calcolo dell'altezza media v;
- Generare un blocco sulla base dei dati contenuti nelle struct `WorldgenData` e posizionarlo all'interno del chunk. Come per la height map, anche qui si vuole fare che ogni thread generi un solo blocco sulla base delle proprie coordinate (x, y, z), che stavolta saranno in tre dimensioni;

- Come già anticipato, la creazione delle decorazioni non verrà parallelizzata in quanto aggiungerebbe molta complessità. In verità è stata del tutto omessa, ma non sarebbe troppo difficile integrarla nel progetto facendo inserire le decorazioni in maniera seriale e sfruttando la height map per rinvenire le coordinate del blocco in superficie su cui sarebbe possibile piazzarle.

Data la memoria limitata a disposizione dei CUDA thread, e nell'ottica di mantenere il codice il più possibile semplice, non si vuole effettuare una traduzione ed un trasferimento di struct complicate come World o Heightmap fra host e device. L'idea è adottare una struct apposita, chiamata `CUDA_RESULT`, in cui inserire man mano tutti i risultati prodotti dalla computazione parallela (Listato 4.1). In questa struct si memorizzerebbero poi dei semplici array, con cui la GPU lavora molto bene.

```
// In cuda/cuda-worldgen.h

// ridefinizione di WorldgenData per codice GPU; notare
   i tipi di dato diversi
typedef struct CudaWorldgenData {
    float h_b;
    long long h, b;
} CUDA_WORLDGEN_DATA;

typedef struct cudaChunkResult {
    int blocks_number;
    // array di tutti i blocchi generati per questo
       chunk
    enum CudaBlockId *blocks;
    // array di tutti i WorldgenData della height map di
        questo chunk
    CUDA_WORLDGEN_DATA *data;
} CUDA_RESULT;
```

Listato 4.1: Struct `CUDA_RESULT`, usata per semplificare le operazioni di trasferimento di informazioni tra host e device.

La struct `CUDA_RESULT` non è altro che la rappresentazione semplificata del contenuto di un chunk, in cui la height map è modellata mediante un array di struct `CUDA_WORLDGEN_DATA` e i blocchi del chunk vengono messi in un array di enum `CudaBlockId`. I `CUDA_WORLDGEN_DATA` sono una ridefinizione per device dei `WorldgenData` originali. Il campo intero `blocks_number` contiene il numero di blocchi generati, in modo da poter arrestare la lettura dell'array `blocks` non appena saranno stati piazzati `blocks_number` blocchi.

4.2 Inserimento di codice in un progetto esistente

Uno dei primi problemi affrontati (oltre all'aver dovuto impostare un ambiente di lavoro su un sistema operativo Unix) è stato capire se fosse possibile integrare il codice CUDA all'interno di un progetto interamente in linguaggio C. CUDA è un'estensione di C++, a sua volta un'estensione di C. Essendo CUDA un super-insieme di C, è evidente che il codice C possa essere chiamato da codice CUDA e compilato mediante nvcc. La cosa meno ovvia è la comunicazione nel senso opposto.

Una prima idea è stata riscrivere tutto il codice all'interno di file .cu, ma questo avrebbe comportato una mole enorme di lavoro e avrebbe impedito di sfruttare il Makefile esistente, che si occupava anche della compilazione delle cinque librerie statiche facenti parte del progetto.

Chiamare codice CUDA da codice C doveva essere possibile in linea di massima, ma con alcune accortezze, specie nel caso in cui si fossero usati meccanismi specifici del C++ come gli oggetti. Come descritto nel Capitolo 2, si possono definire delle funzioni in dei file CUDA/C++ da chiamare poi nel codice C, ma bisogna assicurarsi di scrivere il tutto all'interno di un blocco dichiarato come `extern "C"` {...}. Questa pratica consente di evitare il *name mangling* [8], un processo di modifica dei nomi dati a funzioni e variabili messo in atto dal compilatore C++ che consente di fornire più informazioni al linker in situazioni in cui si fa *overloading* di metodi.

Questa difficoltà è stata risolta studiando più a fondo la struttura del Makefile, cosa che ha permesso di introdurre regole per compilare codice CUDA con nvcc, lasciando inalterata la compilazione con clang dei file .c (Listato 2.2); infine, si è inserito tutto il codice CUDA all'interno dei blocchi `extern "C"`. Un primo test di integrazione è stato effettuato inserendo un programma scorrelato dal progetto dentro ad un file .cu. Si è poi dichiarata una funzione in un file .h, chiamabile dal codice C, che mettesse in esecuzione il codice per GPU contenuto nel file .cu. La chiamata a questo codice è stata inserita nel file main.c, subito prima delle chiamate di inizializzazione della finestra e del loop della finestra. Quando il programma CUDA ha stampato l'output a schermo e poi il gioco programmato da jdah è partito, ho saputo che l'integrazione era andata a buon fine.

I sorgenti CUDA sono stati inseriti in una cartella cuda all'interno di src. I prototipi delle funzioni CUDA sono poi stati inseriti nei file .h già presenti nel progetto originale.

4.3 Traduzione delle struct noise per device

Le struct noise, usate per la tecnica di Perlin noise nella generazione procedurale, devono essere lette quando si crea la height map e quindi devono essere visibili globalmente a tutti i thread.

Questa parte è stata una delle più complicate da mettere in pratica, per una serie di problemi inaspettati descritti in seguito.

4.3.1 Implementazione originale delle struct noise

Nel codice di jdah esiste un tipo generico di struct noise, chiamato `Noise`, che contiene un vettore di byte (char) per mantenere i propri parametri. Queste struct generiche sono specializzate dai tipi `Basic, Octave, Combined, Expscale`. Mentre le struct `Basic` e `Octave` contengono solamente campi numerici, le struct `Combined` contengono due puntatori ad altre generiche struct `Noise` e quelle `Expscale` contengono un puntatore ad una generica struct `Noise`. Questo implica che le struct `Noise` si possono annidare le une nelle altre.

Le struct contengono anche dei puntatori a funzioni `compute()` che variano a seconda del tipo di noise. Ad esempio, al rumore `Basic` corrisponde la funzione `basic_compute()`, e così via (Listato 4.2).

```
// In world/gen/noise.h

// Definizione di un tipo di funzione che restituisce un float (
   f32) e prende come parametri un void *p e tre f32 s, x e z.
typedef f32 (*FNoise)(void *p, f32 s, f32 x, f32 z);

// Struct Noise generica
struct Noise {
    // parametri da dare al "costruttore"
    u8 params[512];
    // funzione "virtual"
    FNoise compute;
};

struct Octave {
    s32 n, o;
};

struct Combined {
    struct Noise *n, *m;
};

struct Basic {
    s32 o;
};

struct ExpScale {
    struct Noise *n;
    f32 exp, scale;
};
```

```
// "Costruttori" di struct noise
struct Noise octave(s32 n, s32 o);
struct Noise combined(struct Noise *n, struct Noise *m);
struct Noise basic(s32 o);
struct Noise expscale(struct Noise *n, f32 exp, f32 scale);
```

Listato 4.2: Definizioni delle struct noise usate da jdah. La struct generica `Noise` contiene un array di byte adibito a mantenere i parametri numerici delle struct specializzate, e un puntatore a funzione compute, di cui si effettua un *override* nelle struct figlie `Basic, Octave, Combined` ed `Expscale`.

Oltre ad essere di difficile lettura, questo fatto comporta delle insidie non da poco, in quanto passare puntatori a funzione a dei CUDA thread, seppure dichiarate come funzioni `__device__`, ha causato errori di accesso a zone di memoria al di fuori del *warp* di cui i thread facevano parte (i *warp* sono gruppi di 32 thread e rappresentano la minima quantità di thread che possono essere schedulati dallo scheduler della GPU per eseguire contemporaneamente), con conseguente crash del programma.

4.3.2 Soluzione: implementazione in CUDA di oggetti Noise

Studiando a lungo il comportamento di questi puntatori a funzione, ho capito che si trattava di un'implementazione del *polimorfismo*, meccanismo tipico dei linguaggi ad oggetti, in C, linguaggio non a oggetti. Per *polimorfismo* si intende, in questo caso, la possibilità di definire metodi con lo stesso nome associati a codice diverso. Qui, il risultato che si vuole ottenere è avere una struct `Noise` *generica* con un metodo *virtual* `compute()`; non ci si vuole curare di specificare manualmente qual è la funzione `compute()` da chiamare a tempo di compilazione; se così fosse, bisognerebbe sapere qual è il tipo di struct noise che si sta usando. Questo è un compito molto difficile in quanto esse si possono innestare le une dentro alle altre in tutte le combinazioni immaginabili. Basti pensare a più struct `Combined` annidate, ciascuna con due puntatori a struct `Noise` generiche.

Possiamo tuttavia sfruttare il fatto che in CUDA sia ammesso utilizzare gli oggetti del C++. In questo modo, il codice sopra si traduce in una serie di classi omonime. La classe padre `Noise` ha un metodo `compute()` virtuale, che le classi figlie implementano a seconda del proprio tipo.

```
// In cuda/noise/cuda-noise.cuh

class CudaNoise {
  public:
    __device__ explicit CudaNoise();
    __device__ virtual ~CudaNoise() {}
    // metodo compute astratto
```

```
    __device__ virtual float compute(float seed, float x
       , float z) = 0;
};

class CudaOctave final : public CudaNoise {
  public:
    __device__ explicit CudaOctave(int n, int o);
    int n, o;
    __device__ ~CudaOctave() override {}
    __device__ float compute(float seed, float x, float
       z) override;
};

class CudaCombined final : public CudaNoise {
  public:
    __device__ explicit CudaCombined(CudaNoise *n,
       CudaNoise *m);
    CudaNoise *n, *m;
    __device__ ~CudaCombined() override {}
    __device__ float compute(float seed, float x, float
       z) override;
};

class CudaBasic final : public CudaNoise {
  public:
    __device__ explicit CudaBasic(int o);
    int o;
    __device__ ~CudaBasic() override {}
    __device__ float compute(float seed, float x, float
       z) override;
};

class CudaExpScale final : public CudaNoise {
  public:
    __device__ CudaExpScale(CudaNoise *n, float exp,
       float scale);
    CudaNoise *n;
    float exp, scale;
    __device__ ~CudaExpScale() override {}
    __device__ float compute(float seed, float x, float
       z) override;
```

```
};
```

Listato 4.3: Classi Noise che sfruttano i meccanismi classici di programmazione ad oggetti in C++.

Una volta creati questi oggetti, nella prima implementazione venivano memorizzati in un array di `Noise`. Questo è stato un errore, in quanto sebbene fosse possibile inserirvi tipi diversi di rumore, tale inizializzazione scorretta ha causato fenomeni di *object slicing*. In C++, l'*object slicing* è la perdita di informazioni che si ha quando in un oggetto di un tipo più generale si memorizza un oggetto di tipo più specifico. Per ovviare a questo problema è stato necessario trasformare l'array di oggetti in un array di puntatori ad oggetto. In questo modo, nell'array vengono inseriti valori di dimensione costante (puntatori) e i veri oggetti sopravvivono intatti da qualche altra parte in memoria.

Il Listato 4.4 mostra come allocare correttamente un array di puntatori sulla GPU, cosa che sulle prime mi ha causato qualche difficoltà.

```
Noise **device_noise_array = nullptr;
const size_t size = n * sizeof(Noise&);

// da notare che il cast si fa comunque a void **, e si
   deve comunque passare il riferimento (&) a
   device_noise_array
cudaMalloc((void **) &device_noise_array, size);
```

Listato 4.4: Allocazione di un array di puntatori sulla GPU. `n` è il numero di oggetti `Noise` desiderati.

4.4 Creazione della height map

Una volta inizializzati gli oggetti `Noise`, si può definire il primo kernel in cui viene calcolata la height map. Ciascun thread ricava l'indice dell'array a cui deve essere inserito il `WorldgenData` da esso creato mediante il seguente algoritmo (gli assi cartesiani possono variare):

```
const unsigned int my_id = blockIdx.x * blockDim.x + threadIdx.x
   ;
const long long my_x = static_cast<long long>(my_id) /
   chunk_size_x;
const long long my_z = static_cast<long long>(my_id) %
   chunk_size_z;
```

Listato 4.5: Algoritmo per determinare le proprie coordinate (x, z) a partire dall'indice globale del thread.

In CUDA, la variabile `my_id` è definita come `unsigned int`, ma le coordinate devono essere trattabili come interi con segno per poter calcolare le posizioni in coordinate globali (e non relative al chunk). Per questo si è reso necessario il cast a `long long`.

Il Listato 4.6 mostra la creazione della heightmap. Una volta che ciascun thread ha calcolato il proprio indice all'interno dell'array `data` passato come parametro, sfrutta gli oggetti `Noise` per calcolare i valori numerici necessari a determinare il bioma di riferimento, e alla fine popola i campi `h_b` e `b` della struct `WorldgenData` all'indice calcolato. Da notare che v, la media delle altezze dei vicini, non viene calcolata in questo kernel: infatti, sebbene esista la direttiva `__syncthreads()` per sincronizzare i CUDA thread, questa funziona solamente per i thread nello stesso blocco della GPU (i blocchi sono unità logiche di raggruppamento di più thread), mentre i vicini potrebbero in realtà essere calcolati da thread in altri blocchi della GPU.

Prima di calcolare v dunque si chiama `cudaDeviceSynchronize()` nel codice dell'host subito dopo la chiamata a questo kernel. Tale funzione assicura che il device abbia finito il proprio lavoro prima che il codice prosegua.

```
__global__ void compute_worldgen_data_gpu(
    CUDA_WORLDGEN_DATA *data,
    [...],
    CudaExpScale **expscales
    ) {

    if (const unsigned int my_id = blockIdx.x * blockDim.x +
       threadIdx.x; my_id < number_of_chunk_columns) {
        const long long world_x = chunk_world_position_x + (
           static_cast<long long>(my_id) / chunk_size_x);
        const long long world_z = chunk_world_position_z + (
           static_cast<long long>(my_id) % chunk_size_z);

        // Esempio di utilizzo degli oggetti compute.
        // expscales e' l'array di puntatori ad oggetti Noise
           passato come parametro al kernel
        float h = expscales[N_H]->compute(world_seed, world_x,
           world_z),
              m, t, r, n, p = [...]

        CudaBiome biome_id = get_biome(h, m, t, n, n + h);
        CudaBiomeData biome = device_biome_data[biome_id];

        [...]

        // Scrittura di h_b e b nel WorldgenData al proprio
           indirizzo
```

```
            data[my_id].h_b = [...]
            data[my_id].b = biome_id;
        }
    }
```

Listato 4.6: Kernel che effettua il calcolo di parte della height map sulla GPU.

Infine, il Listato 4.7 mostra come viene calcolato v per ogni elemento della mappa, in un kernel apposito chiamato `stencil_compute_v()`.

```
    __global__ void stencil_compute_v(
        CUDA_WORLDGEN_DATA *data,
        const int chunk_size_x,
        const int chunk_size_z
        ) {
        const unsigned int my_id = blockIdx.x * blockDim.x +
            threadIdx.x;
        const int my_x = static_cast<int>(my_id) / chunk_size_x;
        const int my_z = static_cast<int>(my_id) % chunk_size_x;
        assert(my_x < chunk_size_x);
        const int down_left = get_index_from_coordinates(my_x - 1,
            my_z - 1, chunk_size_x, chunk_size_z);
        const int up_left = get_index_from_coordinates(my_x + 1,
            my_z - 1, chunk_size_x, chunk_size_z);
        const int down_right = get_index_from_coordinates(my_x - 1,
            my_z + 1, chunk_size_x, chunk_size_z);
        const int up_right = get_index_from_coordinates(my_x + 1,
            my_z + 1, chunk_size_x, chunk_size_z);

        float v = 0.0f;
        v += data[down_left].h_b;
        v += data[down_right].h_b;
        v += data[up_left].h_b;
        v += data[up_right].h_b;
        // Calcolo della media aritmetica
        v *= 0.25f;
        assert(!isnan(v));

        // Inserimento dell'altezza media nell'array data
        data[get_index_from_coordinates(my_x, my_z, chunk_size_x,
            chunk_size_z)].h = v;
    }
```

Listato 4.7: Kernel che effettua il calcolo della media delle altezze delle colonne della height map, per completarne la generazione. I thread sono stati sincronizzati mediante una chiamata a `cudaDeviceSynchronize()` subito dopo la chiamata al kernel `compute_worldgen_data_gpu()` (Listato 4.6).

4.5 Generazione dei blocchi

Con la height map fatta, non resta che creare i blocchi che popoleranno il chunk. Anche in questo caso si adopera un kernel, `generate_blocks_gpu()`, che prende come parametro un array di enum BlockId dove ciascun thread metterà il proprio blocco, generato nel rispetto dei dati contenuti nella mappa altimetrica.

```
1     __global__ void generate_blocks_gpu(
2         CUDA_WORLDGEN_DATA *data,
3         CudaBlockId *blocks,
4         [...],
5         int *array_of_partial_results
6     ) {
7         /* Ogni thread all'interno di uno stesso blocco modifica il
              proprio valore dentro questo array; alla fine, tutti i
              thread di uno stesso blocco compiono una reduction sull'
              array, per calcolare la somma di blocchi generati.*/
8         __shared__ int local_generated_blocks[BLKDIM];
9         const unsigned int lindex = threadIdx.x; // local index
10        local_generated_blocks[lindex] = 0;
11        const long long global_index = blockIdx.x * blockDim.x +
              threadIdx.x;
12
13        if (global_index < total_blocks_number) {
14            // Calcolo delle coordinate (x, y, z) a partire dal
                  global_index
15            const long long my_y = global_index % chunk_size_x;
16            const long long my_x = global_index / (chunk_size_y *
                  chunk_size_z);
17            const long long completed_yz_planes_elements = my_x *
                  chunk_size_x * chunk_size_z;
18            const long long my_z = (global_index -
                  completed_yz_planes_elements) / chunk_size_y;
19            const long long my_xz = my_x * chunk_size_x + my_z;
20
21            const CUDA_WORLDGEN_DATA my_data = data[my_xz];
```

```
22              const long long h = my_data.h;
23              const auto biome = static_cast<CudaBiome>(my_data.b);
24
25              [...]
26
27              const CudaBiomeData biome_data = device_biome_data[biome
                    ];
28              const CudaBlockId top_block = h > 48 ? CUDA_SNOW :
                    biome_data.top_block,
29                                  under_block = biome_data.bottom_block;
30
31              const long y_w = chunk_world_position_y + static_cast<
                    long>(my_y);
32              CudaBlockId block = CUDA_AIR;
33
34              if (y_w > h && y_w <= WATER_LEVEL) {
35                  block = CUDA_WATER;
36              } else if (y_w > h) {
37                  block = CUDA_AIR;
38                  /* Se il thread genera un blocco d'aria, in realta'
                       non deve essere preso in considerazione per il
                       conteggio dei blocchi totali. */
39                  local_generated_blocks[lindex]--;
40              } else if (y_w == h) {
41                  block = top_block;
42              } else if (y_w >= (h - 3)) {
43                  block = under_block;
44              } else {
45                  block = CUDA_STONE;
46              }
47
48              blocks[global_index] = block;
49              local_generated_blocks[lindex]++;
50              __syncthreads();
51              /* Alla fine, i thread di tutti i blocchi GPU calcolano
                   una riduziona parziale sul proprio array shared, che
                   contiene i blocchi (del gioco) generati da ciascun
                   thread. Questo algoritmo funziona perche' il numero
                   di thread per blocco e' una potenza di 2 (512 thread
                   per blocco = 2^9). */
52              int bsize = blockDim.x / 2;
53              while (bsize > 0) {
54                  if (lindex < bsize) {
```

```
55                      local_generated_blocks[lindex] +=
                            local_generated_blocks[lindex + bsize];
56                  }
57                  bsize = bsize / 2;
58                  __syncthreads();
59              }
60              if (0 == lindex) {
61                  array_of_partial_results[blockIdx.x] =
                        local_generated_blocks[lindex];
62              }
63          }
64      }
```

Listato 4.8: Kernel per la generazione dei blocchi all'interno di un chunk.

Il Listato 4.8 mostra il kernel per la generazione dei blocchi. Questo kernel si suddivide in due parti principali: la prima è quella che determina il blocco da mettere nella posizione (x, y, z) del thread corrente, e la seconda è quella che calcola il numero di blocchi totali generati.

La prima parte non presenta differenze particolari rispetto all'implementazione seriale; tutta la complessità è data dall'algoritmo usato per ricavare le coordinate del blocco a partire dall'indice globale del thread. Significativo è l'`else if` a riga 36: mentre nel codice originale quando il ciclo raggiungeva un'altezza `y_w` maggiore dell'altezza `h` specificata dalla height map si saltava all'iterazione successiva con un'istruzione `continue`, nel codice CUDA non è possibile inserire questa istruzione, per via dell'assenza di un ciclo. Inizialmente si era pensato di arrestare l'esecuzione del thread con un'istruzione `return`; ma questo comportava due problemi:

- nell'array `data` non veniva impostato alcun blocco, per cui ci si ritrovava un blocco "casuale" dato dalla memoria allocata con `cudaMalloc()` che non veniva inizializzata;
- le istruzioni `__syncthreads()` successive provocavano comportamenti indefiniti, in quanto per funzionare tali chiamate a funzione devono essere eseguite da **tutti i thread dello stesso blocco GPU** [7].

L'alternativa applicata è impostare il blocco come un blocco d'aria e non conteggiarlo per il calcolo dei blocchi totali generati.

La seconda parte del kernel si occupa per l'appunto di calcolare il numero di blocchi del gioco creati dal proprio blocco GPU di thread. Questo conto richiede di sapere quanti blocchi ha piazzato ciascun thread (se 0 o 1), e quindi necessita di sincronizzazione a livello locale del blocco GPU.

Ciascun blocco GPU ha a disposizione una memoria *shared* locale, visibile soltanto ai thread dello stesso blocco. Gli accessi a questo tipo di memoria sono molto efficienti e

veloci. Per dichiarare una variabile condivisa, si antepone la keyword `__shared__` al nome della variabile. Nel caso in esame si è usato un array shared di interi. Ciascun thread del blocco calcola il proprio indice locale in questo array di interi, e lo inizializza a 0. Se viene generato un blocco diverso dall'aria, questo valore verrà incrementato ad 1. Per sapere il numero di blocchi del gioco piazzati dai thread in un blocco GPU, si effettua una somma di tutti i numeri contenuti in questo array, chiamato `local_generated_blocks` (riga 8).

In letteratura, sommare gli elementi di un array in un contesto di esecuzione parallela è un esempio di un pattern di programmazione parallela chiamato *reduction*. Si ha una *reduction* ogni volta che a tutti gli elementi di un array deve essere applicato un operatore associativo (come la somma) e il risultato finale dev'essere un valore scalare.

Il modo in cui questa somma è effettuata è il seguente: ciascun blocco GPU partiziona l'array in due metà. I thread della prima metà calcolano una somma fra il numero al proprio indice e il numero all'indice corrispondente dell'altra metà dell'array. Questo processo viene ripetuto dimezzando ogni volta la porzione di array considerata, fino a quando la somma finale si troverà all'indice 0 dell'array (Figura 4.1).

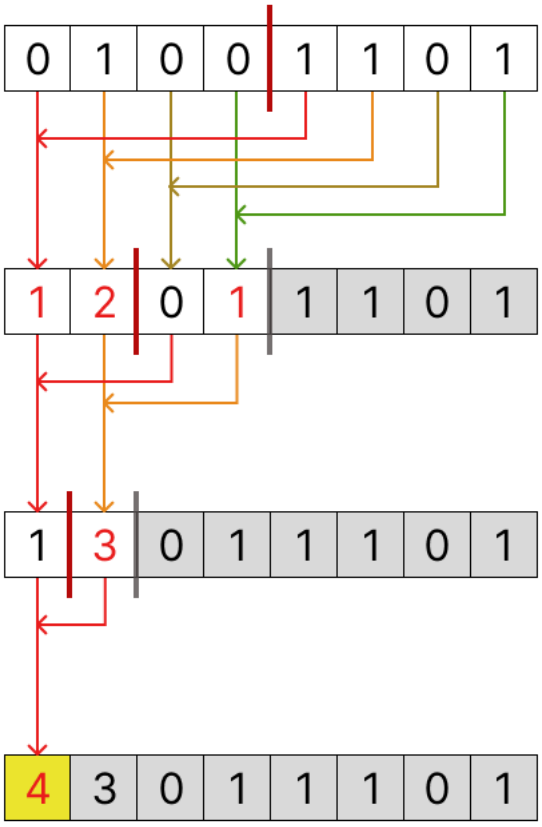

Figura 4.1: Esempio di reduction sulla GPU. A partire da un array di dimensione pari ad una potenza di due, i vari thread calcolano delle somme parziali fra il proprio indice e quello corrispondente dell'altra metà in cui si partiziona l'array. Ad ogni passaggio si dimezza la porzione di array su cui si lavora. Alla fine, all'indice 0 sarà presente la somma completa degli elementi dell'array.

A questo punto, un thread qualsiasi per ogni blocco (convenzionalmente il thread 0) si occupa di mettere la somma parziale nell'array `array_of_partial_results` passato come parametro al kernel (riga 61). La CPU effettuerà una somma finale delle somme parziali calcolate da ciascun blocco GPU, ottenendo così il numero di blocchi del gioco generati in tutto in questo chunk. Questo numero verrà poi usato nel codice host, così non appena si saranno piazzati quel numero di blocchi, si potrà interrompere in anticipo il ciclo che altrimenti passerebbe in rassegna sempre tutti i 32 768 (32^3) blocchi di un chunk.

4.6 Esempi di mondi generati

Seguono le immagini di alcuni panorami generati dall'algoritmo parallelizzato. Si notano le transizioni graduali da un bioma all'altro, che in genere non fanno sospettare la suddivisione in chunk dell'ambiente di gioco. Questo è un vantaggio dato dal fatto che le uniche discriminanti per decidere il tipo di ciascun blocco sono la height map e la posizione in coordinate globali di tale blocco.

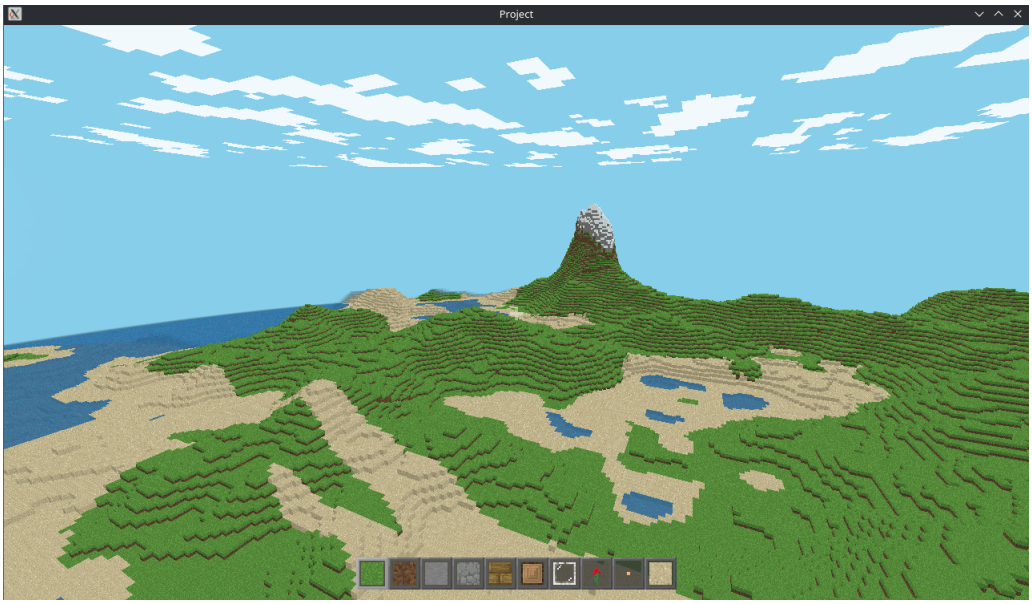

Figura 4.2: Prateria con spiagge e laghetti.

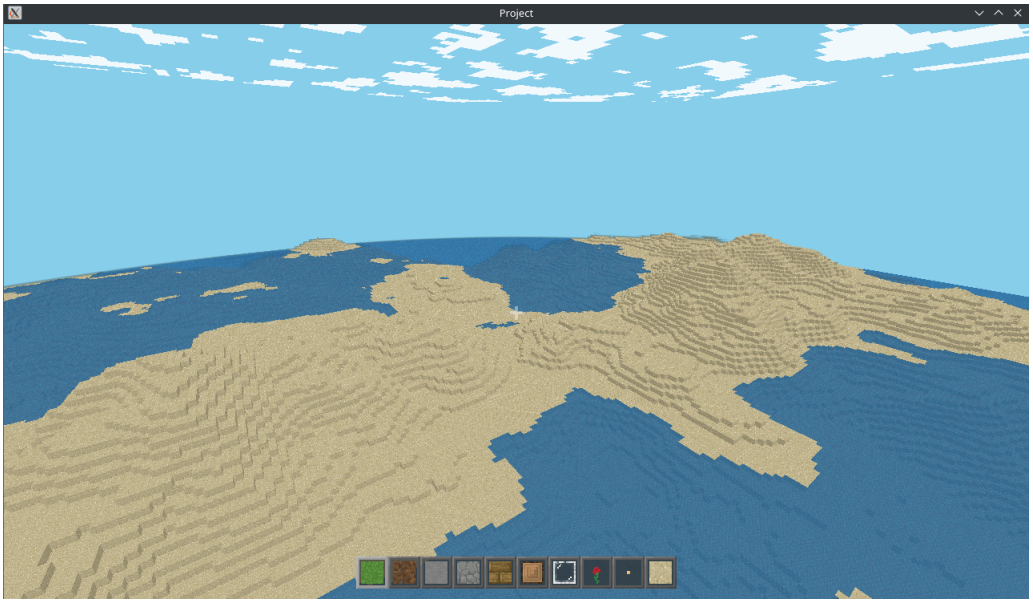

Figura 4.3: Spiaggia nei pressi di un oceano.

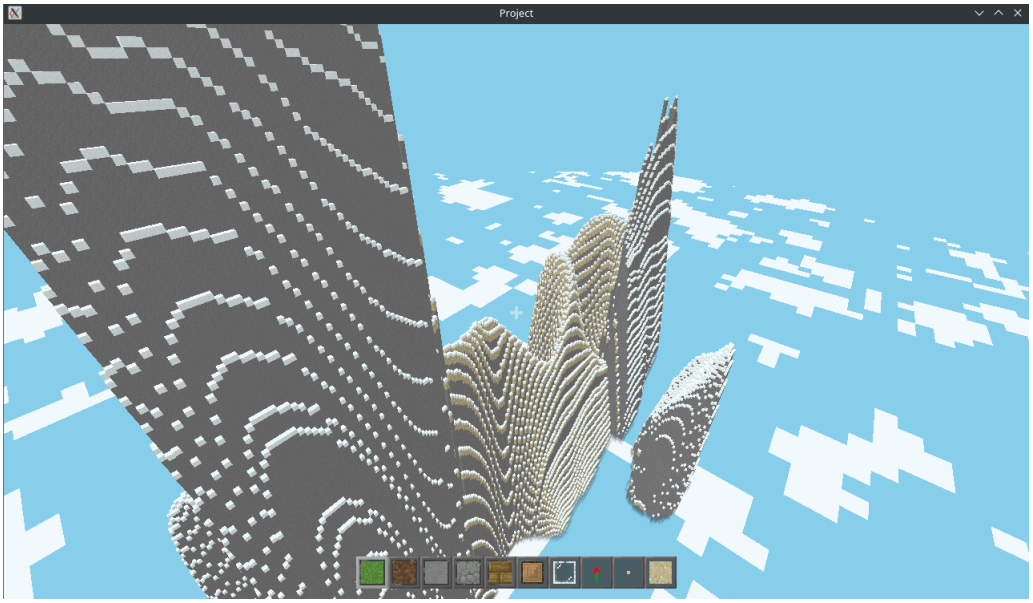

Figura 4.4: Montagna immensa, vista dall'alto.

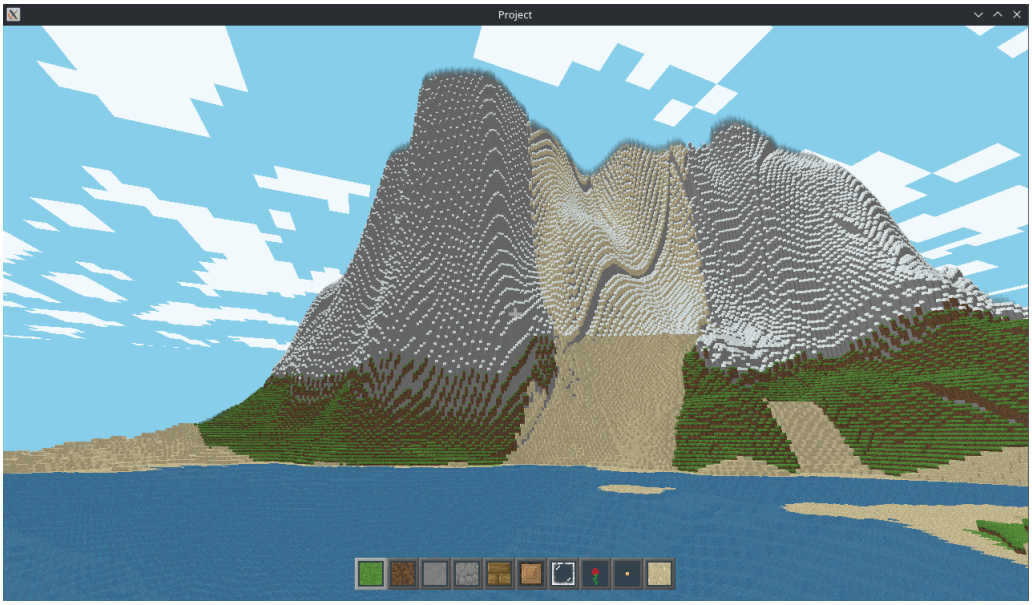

Figura 4.5: Montagna immensa, vista dal basso.

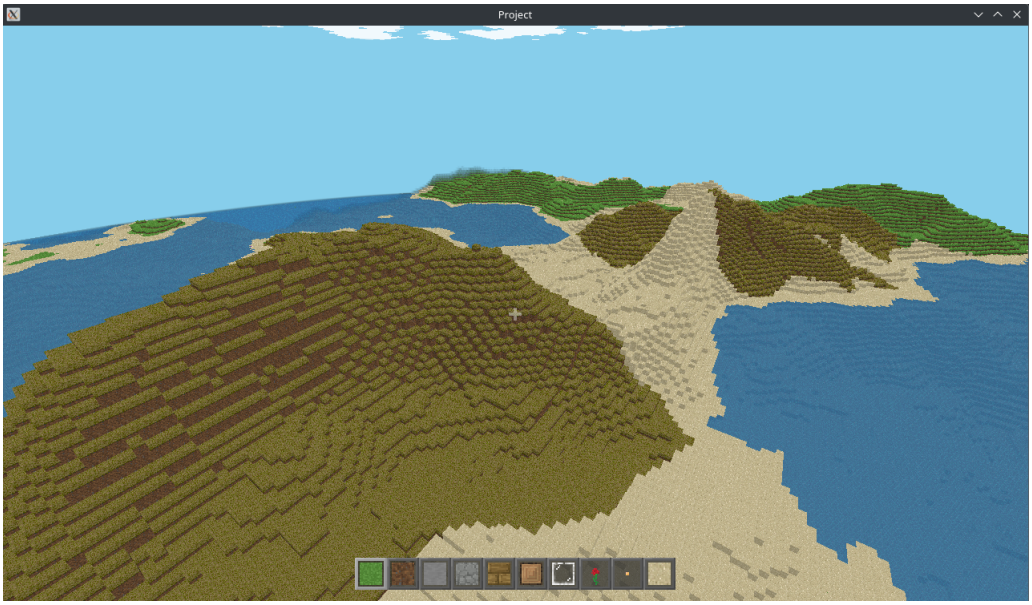

Figura 4.6: Isole con savane, spiagge e praterie.

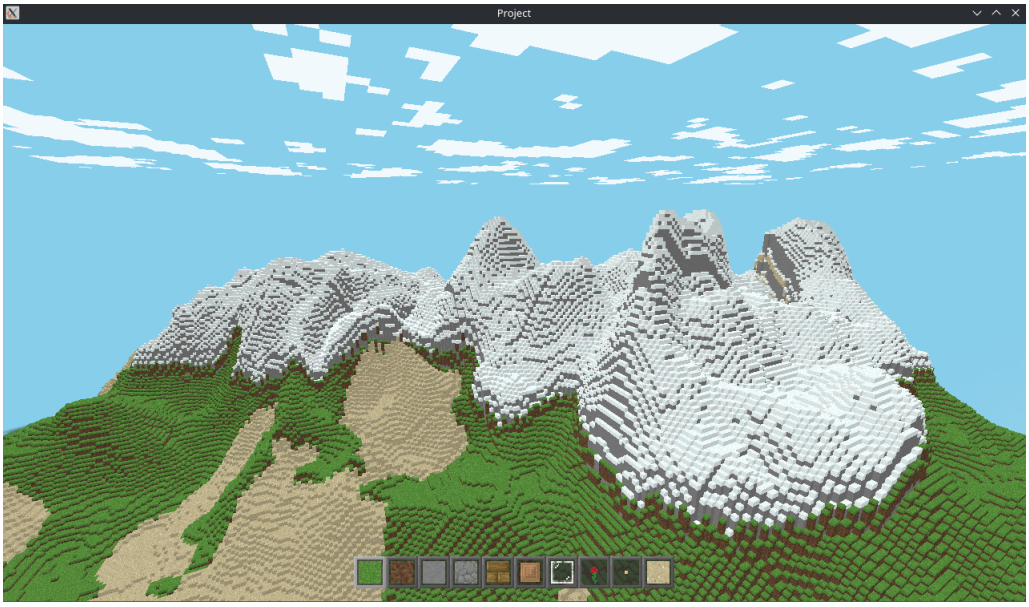

Figura 4.7: Catena montuosa.

Capitolo 5

Analisi delle prestazioni

5.1 Raccolta dei dati

Normalmente per valutare l'efficienza di un programma parallelizzato sulla GPU se ne deve considerare un'implementazione efficiente (ossia parallelizzata su CPU) e bisogna confrontare i tempi di esecuzione in entrambi i casi. Le misure si dovrebbero poi effettuare variando il *throughput*, ossia la quantità di lavoro totale da svolgere, e studiando il comportamento delle due implementazioni.

In questo caso si paragonerà la versione parallela direttamente alla versione seriale così com'era stata ideata da jdah. Entrambi i progetti verranno compilati abilitando il livello massimo di ottimizzazione (opzione `-O3` dei compilatori) e rimuovendo i simboli di debug (opzioni `-g` e `-G`).

Si effettueranno delle misure ripetute per verificare il tempo di generazione di un singolo chunk, di cui poi si farà una media per ridurre il più possibile l'errore. Una volta fatto ciò, si confronterà il tempo impiegato a generare tutti i chunk del mondo, al variare del numero totale di chunk da allocare contemporaneamente. In entrambi i casi, si lascerà eseguire ciascun programma per un totale di 10 frame. Si garantisce che durante la generazione non vengano soltanto piazzati i blocchi, ma anche create le height map, almeno qualche volta.

5.2 Misure ed interpretazione dei dati

5.2.1 Generazione di un singolo chunk

La Tabella 5.1 confronta i tempi medi impiegati per la generazione di un solo chunk dalla versione seriale e quella CUDA. La generazione si suddivide nella creazione di 32^2 `WorldgenData` all'interno della height map e un massimo di 32^3 blocchi da piazzare nel chunk. Il numero di chunk totali presenti nel mondo contemporaneamente è stato

lasciato a 16, come da impostazioni predefinite; ciò implica una buona probabilità che si generino delle height map più di una volta nel corso del calcolo dei 10 frame.

Versione	**Tempo 1**	**Tempo 2**	**Tempo 3**	**Tempo 4**	**Tempo 5**	**Media**
Seriale	1.81 ms	1.52 ms	1.77 ms	1.51 ms	1.52 ms	**1.63 ms**
CUDA	0.26 ms	0.28 ms	0.26 ms	0.27 ms	0.29 ms	**0.27 ms**

Tabella 5.1: Confronto dei tempi medi impiegati per la generazione di un singolo chunk.

La versione CUDA genera un solo chunk in media 6 volte più velocemente della versione originale.

5.2.2 Generazione di tutti i chunk del mondo

Seguono i tempi medi impiegati dalle due versioni per la generazione di tutti i chunk nel mondo, in numero variabile, per studiare il comportamento dei programmi all'aumentare del *throughput*. Anche in questo caso, il numero di `WorldgenData` per ciascuna height map è 32^2 e il numero massimo di blocchi che vengono piazzati in un chunk è 32^3.

Versione seriale						
Chunk totali	**T1**	**T2**	**T3**	**T4**	**T5**	**Media**
16	5.42 ms	3.89 ms	4.16 ms	5.42 ms	3.61 ms	**4.50 ms**
24	4.56 ms	5.90 ms	6.04 ms	5.01 ms	5.72 ms	**5.45 ms**
32	6.32 ms	7.06 ms	7.06 ms	6.09 ms	6.02 ms	**6.51 ms**
40	11.33 ms	9.49 ms	10.25 ms	9.46 ms	9.46 ms	**10.00 ms**
48	17.10 ms	14.48 ms	14.29 ms	14.24 ms	14.29 ms	**14.88 ms**
56	22.97 ms	24.03 ms	21.65 ms	21.66 ms	23.62 ms	**22.79 ms**

Tabella 5.2: Tempo medio impiegato dalla versione seriale a processare tutti i chunk presenti nel mondo.

Versione CUDA							
Chunk	**T1**	**T2**	**T3**	**T4**	**T5**	**Media**	**Speedup**
16	3.24 ms	2.05 ms	3.16 ms	2.05 ms	1.98 ms	**2.50 ms**	**1.80**
24	2.94 ms	3.91 ms	2.81 ms	3.42 ms	2.92 ms	**3.20 ms**	**1.70**
32	4.74 ms	4.61 ms	5.31 ms	4.38 ms	4.82 ms	**4.77 ms**	**1.36**
40	8.18 ms	8.12 ms	7.81 ms	7.90 ms	7.84 ms	**7.97 ms**	**1.25**
48	15.45 ms	12.52 ms	13.85 ms	12.68 ms	13.41 ms	**13.58 ms**	**1.10**
56	20.03 ms	19.81 ms	20.49 ms	20.30 ms	20.21 ms	**20.17 ms**	**1.13**

Tabella 5.3: Tempo medio impiegato dalla versione CUDA a processare tutti i chunk presenti nel mondo.

La Tabella 5.2 mostra i tempi di generazione procedurale di tutti i chunk del mondo ad ogni frame, al variare del numero di chunk, per la versione seriale del programma; la Tabella 5.3 mostra i tempi per la versione parallela. Nonostante in questo caso il tempo medio impiegato dalle due versioni non si discosti troppo, si nota comunque un vantaggio della versione parallelizzata su GPU, che arriva ad avere uno *speedup* di fino a 1.80 (il che significa che è più veloce dell'80%). Lo *speedup* è definito come $\frac{T_s}{T_p}$, dove T_s è il tempo impiegato dalla versione seriale e T_p è il tempo impiegato dalla versione parallela.

La Figura 5.1 mostra l'andamento dei tempi di generazione procedurale dei chunk del mondo all'aumentare del numero di chunk. Si nota che la versione CUDA impiega sempre meno tempo rispetto a quella seriale. Entrambe le versioni tuttavia richiedono più di $16.\overline{6}$ ms quando il numero di chunk si alza a 56, il che implica la necessità di adottare accortezze per poter mantenere un framerate fluido di 60 FPS, se si volessero tenere contemporaneamente tutti quei chunk.

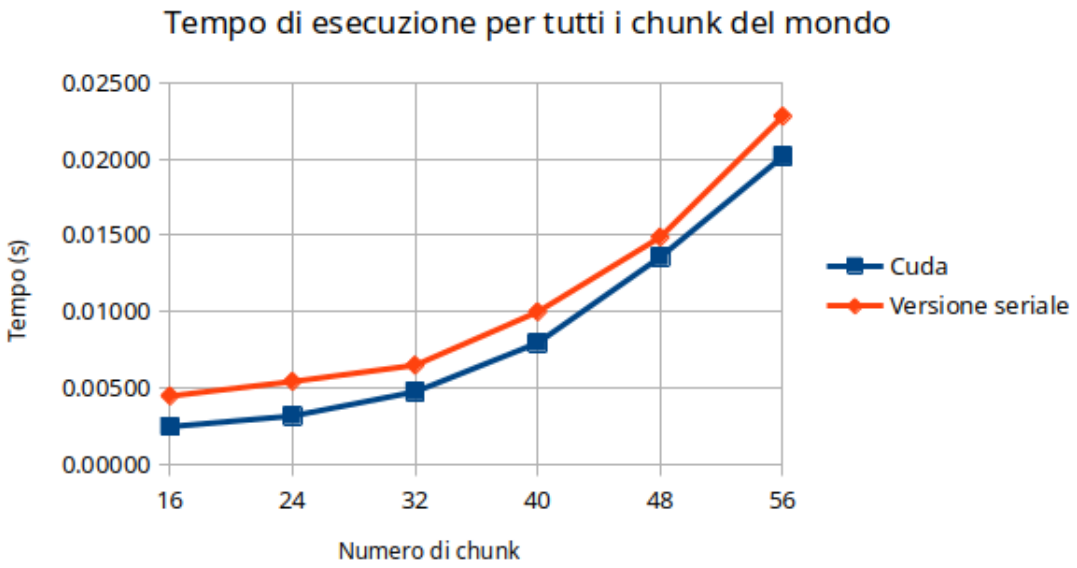

Figura 5.1: Andamento dei tempi di esecuzione della versione CUDA (in blu) e di quella seriale (in rosso) all'aumentare del numero di chunk nel mondo.

Capitolo 6

Conclusioni

6.1 Risultati

L'applicazione di tecniche di *high performance computing* al progetto di jdah ha prodotto dei miglioramenti nei tempi d'esecuzione, confermando l'ipotesi che i videogiochi, in cui spesso ci sono vincoli molto stringenti di efficienza, beneficino in alcuni casi da interventi di parallelizzazione sulla GPU. L'identificazione di porzioni di codice seriale che svolgessero compiti ripetitivi e semplici su grandi quantità di dati in maniera indipendente fra le varie iterazioni è stata un utile indizio per comprendere in quale punto del codice intervenire: nello specifico, si è modificata la generazione procedurale della height map e il processo di inserimento dei blocchi all'interno di ciascun chunk, adattandoli al paradigma CUDA; ciò ha portato a discreti miglioramenti nei tempi di esecuzione anche con una semplice prima versione dell'implementazione. Il percorso ha comportato diverse sfide, superate mediante la familiarizzazione con gli strumenti per lo sviluppo e il debugging delle applicazioni CUDA e la creazione di piccoli prototipi ogni volta che ce ne fosse stato il bisogno, oltre che con l'approfondimento delle conoscenze in materia di compilazione e linking separati, linguaggi C e C++ e lo studio di un progetto di dimensioni considerevoli.

6.2 Sviluppi futuri

Il progetto può essere ulteriormente espanso e raffinato. Mentre è certamente possibile aggiungere funzionalità quali la scrittura delle modifiche su file per conservarle anche quando si lascia il chunk corrente, introdurre elementi di gioco effettivo o correggere i bug presenti, rimangono tuttavia anche altri possibili interventi in ambito di ottimizzazione che possono essere fatti sia per produrre risultati identici a quelli del progetto originale, sia per apportare ulteriori miglioramenti:

- aggiungere il codice per la generazione delle decorazioni quali alberi, fiori e cespugli, valutando se sia applicabile il paradigma CUDA o se convenga effettuare tale operazione sulla CPU;
- aggregare la generazione procedurale di più chunk contemporaneamente, invece che parallelizzare la generazione di un solo chunk alla volta;
- inserire la possibilità di variare dinamicamente il numero di chunk da caricare nel mondo a tempo di esecuzione (funzionalità presente nel gioco Minecraft originale, e chiamata *render distance*);
- migliorare gli algoritmi adoperati per la generazione delle montagne, che per quanto interessanti hanno degli aspetti che ricordano molto delle funzioni matematiche più che delle strutture davvero simili a quelle naturali;
- generalizzare il codice per introdurre caverne o applicare altri algoritmi di generazione procedurale, come la simulazione dell'erosione.

Ringraziamenti

3 Novembre 2024

Scrivere i ringraziamenti è in assoluto la mia parte preferita dello scrivere un'opera, perché significa che tale opera è finalmente conclusa. Finora però non mi era ancora capitato di concludere anche un capitolo della mia vita assieme all'opera, e penso che questo renda il momento ancora più importante.

Nel corso di questi tre anni e mezzo di studi sembra che i tempi di esecuzione si siano abbreviati anche nella vita reale: sono passato dall'essere un ingenuo che non sapeva scrivere due righe di codice senza far crashare il computer solo l'altro giorno, a riuscire ad affrontare e a scavare via manciate di conoscenza ed esperienza da progetti che finalmente si avvicinano a ciò che ho sognato di fare fin dalla quarta elementare (dodici anni fa): creare videogiochi. Poter prendere in prestito gli strumenti dei narratori, degli artisti, degli architetti, degli ingegneri, di Dio, e plasmare un'immagine virtuale della realtà in cui far accadere la magia delle storie ben raccontate è una prospettiva che mi ha stregato fin dalle mie prime interazioni con queste opere; riuscire a muovere la mia matita tremolante su un foglio e a buttare giù qualche sorgente che mi permetta di fare i miei primi passi in questa direzione ha per me un valore inestimabile. In questi anni ho studiato, esplorato, divorato ambiti del sapere che mi hanno fatto meravigliare e desiderare di capire più a fondo i misteri che si celano in tutto, sia nella matematica e nella fisica, che nelle macchine e nel codice, che nei giochi e nei software che compiono quelli che per me sono piccoli miracoli.

Ma lungi da me trarre conforto e apprezzamento solo dalla mera cultura che questi anni mi hanno lasciato addosso. Vedere le spunte verdi dei test che passano o affrontare un limite notevole con Taylor sono soddisfazioni che seppur di valore, non sempre fanno parte dei ricordi base che vorremmo incorniciare e ci farebbero arrestare anche fra anni per ammirarli per qualche attimo, con un fugace sorriso che si abbozza agli angoli della bocca, malinconico o gioioso che sia.

Quei momenti-quadro che ho collezionato in questi anni sono stati possibili soltanto grazie alle persone meravigliose che mi hanno circondato. Ho dovuto affrontare momenti difficili, momenti in cui ho ridiscusso tutte le mie certezze e in cui la mia determinazione ha vacillato. È stato allora che la loro vicinanza è stata fondamentale, ed è stato allora che ho imparato ad apprezzarli dal profondo del cuore, al punto di dirmi fortunatissimo

persino in quei mesi in cui non vedevo splendere alcuna luce. Non mi bastano le pareti della reggia di Versailles per appendere tutti i quadri che riguardano queste persone. Vediamo se queste pagine mi basteranno per ringraziarli, almeno in minima parte.

Ringrazio la mia famiglia; mio padre per le parole severe di conforto che mi hanno spronato a uscire dai vicoli bui in cui mi ero cacciato, e per quelle poche parole di lode che escono fuori soltanto nei momenti davvero importanti, rendendoli eterni; mia sorella per le fervide discussioni sulla letteratura, libri, film e serie animate, e per i momenti di confronto profondo con l'adolescente taciturna ma sempre in osservazione che è diventata; mia madre per avermi buttato in faccia i miei limiti e avermi chiesto di superarli per realizzarmi e coronare il mio sogno, ora che sono ad un soffio dall'afferrarlo.

Ringrazio tutti i miei amici per un'infinità di cose di cui menzionerò solo alcune; per le risate, le torte, i giochi, le uscite, le campagne di Dungeons & Dragons, i film, le pizze, le prese in giro, le lacrime, le serate in chiamata su Discord a lamentarci dei problemi vari dell'esistenza, il fatto di avermi mostrato che non sono da solo. Ringrazio i membri dell'associazione S.P.R.I.Te. per i gesti di bene gratuito che hanno compiuto nella loro opera di volontariato, e anche per avermi tirato fuori dal mio buco sottoterra esponendomi a seminari, viaggi, professori, istituzioni, discorsi pubblici e alla dannata burocrazia.

Ringrazio i compagni di parkour che volano in giro per la palestra facendomi dubitare di meritarmi il voto preso in fisica; volare con loro è uno dei pochi momenti in cui la mia mente si libera dalle incombenze e dalle preoccupazioni.

Ringrazio i miei professori, che con le loro passioni e controversie hanno colorato e arricchito di molto la mia esperienza universitaria. A coloro che hanno creduto nella bellezza di ciò che insegnavano devo molta della mia sicurezza e passione attuale. In questa categoria rientra anche il mio relatore, il professor Marzolla, a cui rivolgo un sentito grazie per avermi lasciato libero di sperimentare con ciò che chiamo la mia vocazione e per la meticolosità degli aiuti che mi ha fornito e delle lezioni che ha svolto sia nel corso di Algoritmi che in quello di High Performance Computing. Confido nel fatto che ciò che ho imparato con lui mi tornerà utile quando dovrò occuparmi di rendere i giochi un'esperienza giocabile e abitabile per tutti.

Questo momento non sarebbe possibile senza tutti loro.

Bibliografia

[1] Automatic Variables (GNU make) — gnu.org. https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html#Automatic-Variables. [Accessed 29-10-2024].

[2] Understanding Perlin Noise — adrianb.io. https://adrianb.io/2014/08/09/perlinnoise.html. [Accessed 27-10-2024].

[3] Saneef H. Ansari. Noise Octaves — saneef. https://observablehq.com/@saneef/noise-octaves. [Accessed 27-10-2024].

[4] L V Bogachev. Random walks in random environments. *arXiv.org*, 2007.

[5] Green Dale. *Procedural Content Generation for C++ Game Development : Get to Know Techniques and Approaches to Procedurally Generate Game Content in C++ Using Simple and Fast Multimedia Library.* Community Experience Distilled. Packt Publishing, 2016.

[6] Ketra Games. Creating Terrain from Heightmaps - Unity Game Development Tutorial — ketra-games.com. https://www.ketra-games.com/2021/08/creating-terrain-from-heightmaps-unity-game-tutorial.html. [Accessed 27-10-2024].

[7] Mark Harris. Using Shared Memory in CUDA C/C++ — NVIDIA Technical Blog — developer.nvidia.com. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/, 2013. [Accessed 01-11-2024].

[8] IBM. Name mangling (C++ only) — ibm.com. https://www.ibm.com/docs/en/i/7.5?topic=linkage-name-mangling-c-only. [Accessed 31-10-2024].

[9] Andrew Ilachinski. *Cellular Automata: A Discrete Universe.* World Scientific Publishing, Singapore, Singapore, July 2001.

[10] Eugene Izhikevich, John Conway, and Anil Seth. Game of life. *Scholarpedia journal*, 10(6):1816, 2015.

[11] JetBrains. CUDA projects — CLion — jetbrains.com. https://www.jetbrains.com/help/clion/cuda-projects.html#cuda-gdb. [Accessed 30-10-2024].

[12] Wanwan Li. Synthesizing realistic cracked terrain for virtual arid environment generation. In *2023 8th International Conference on Communication, Image and Signal Processing (CCISP)*, pages 361–366, 2023.

[13] Steven Lisberger. Tron, 1982.

[14] NVIDIA. CUDA C++ Programming Guide — docs.nvidia.com. https://docs.nvidia.com/cuda/cuda-c-programming-guide/. [Accessed 28-10-2024].

[15] Robert Nystrom. *Game Programming Patterns*. Genever Benning, November 2014. pp. 123–137.

[16] Leonardo Tortoro Pereira, Paulo Victor de Souza Prado, Rafael Miranda Lopes, and Claudio Fabiano Motta Toledo. Procedural generation of dungeons' maps and locked-door missions through an evolutionary algorithm validated with players. *Expert Systems with Applications*, 180:115009, 2021.

[17] Ken Perlin. Improving noise. *ACM transactions on graphics*, 21(3):681–682, 2002.

[18] Aitor Santamaría-Ibirika, Xabier Cantero, Sergio Huerta, Igor Santos, and Pablo G. Bringas. Procedural playable cave systems based on voronoi diagram and delaunay triangulation. In *2014 International Conference on Cyberworlds*, pages 15–22, 2014.