Perlin Noise Generator in CUDA

~

Luigi Rapetta

Indice

1. Descrizione del problema

Col presente progetto, si propone un generatore di Perlin noise bidimensionale, progettato per sfruttare gli stream multipli e avente diversi parametri di configurazione per la generazione del rumore.

I seguenti paragrafi espongono brevemente la procedura del Perlin noise e, infine, discutono gli obiettivi posti per la realizzazione del generatore.

1.1. Il Perlin noise

Il Perlin noise è una funzione per la generazione di rumore sviluppata da Ken Perlin. Esso è impiegato per la generazione procedurale di height-map, texture e terreni. È comunemente implementato per generazione di rumore monodimensionale, bidimensionale e tridimensionale, anche se è possibile implementarlo per un numero arbitrario di dimensioni.

Per i fini legati al progetto, si analizza il caso bidimensionale.

Il Perlin noise è caratterizzato dal fatto che un valore di un pixel della mappa è correlato con i valori del proprio vicinato. Il valore associato a un pixel, in un contesto bidimensionale, può essere definito *altezza*.

Le altezze sono calcolate con l'utilizzo di un *lattice* di *gradienti* (Fig. 1), di risoluzione minore rispetto a quella della mappa che si desidera generare. I gradienti sono vettori bidimensionali generati randomicamente, con coordinate (x, y) che ricadono nel range [-1.0, 1.0], aventi come origine gli angoli delle celle del lattice.

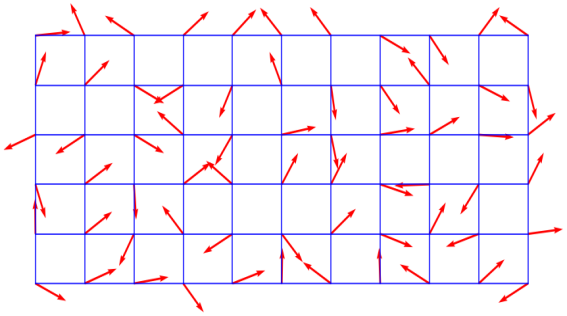

Fig. 1: lattice di gradienti {1}; notare come i gradienti siano disposti lungo gli angoli del lattice.

Concettualmente, il gradiente modella uno *slope*[1]. Il verso e la direzione determinano dove tende a innalzarsi; il magnitudo ne indica la ripidità.

1 Da intendersi come "inclinazione", alludendo al concetto di altezza.

Il lattice, nonostante sia a una risoluzione più bassa rispetto alla mappa del rumore, deve essere interpretato come se fosse completamente mappato sulla mappa del rumore: ad ogni punto del lattice, corrisponde un pixel sulla mappa del rumore.

Per ogni pixel P sulla mappa di risoluzione maggiore vi sono quattro gradienti sul lattice che si avvicinano il più possibile a P (a sinistra di Fig. 2). Tra le origini dei quattro gradienti e P vengono tracciati dei vettori di distanza (a destra di Fig. 2).

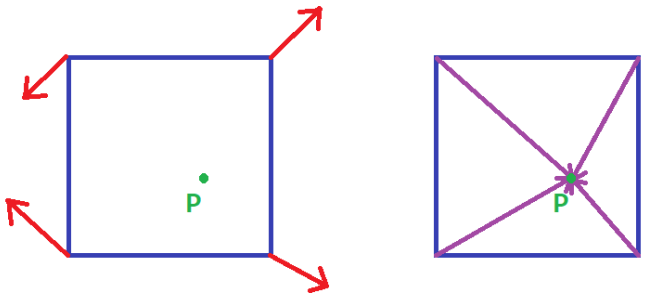

Fig. 2: a sinistra, i gradienti (rosso) più vicini a P; a destra, i vettori (viola) dall'origine del gradiente a P.

Viene calcolato, per ogni gradiente, un valore di altezza associabile a P. Esso equivale al prodotto scalare tra il gradiente e il relativo vettore distanza. Dopodiché, le quattro altezze risultanti devono essere sottoposte a un'interpolazione, nella quale i valori relativi a gradienti più vicini a P ne influenzano maggiormente il risultato. L'altezza interpolata è il valore di altezza finale associato a P.

1.2. Rumore frattale

Il Perlin noise è una funzione che può essere sommata con sé stessa diverse volte per generare rumore frattale. Ogni iterazione di una funzione di noise è detta *ottava*.

Ad ogni ottava si raddoppia la frequenza del rumore (perciò, la densità del lattice di gradienti), e il loro peso aumenta o diminuisce per un fattore moltiplicativo di *falloff*.

La generazione di rumore frattale può essere descritta in pseudo-codice come:

```
f ← initial frequency
weight ← 1.0
for each octave do:
          map ← map + noise(f) * weight
          f ← 2 * f;
          weight ← weight * falloff
```

Se il falloff corrisponde a 0.5, allora il rumore generato è detto *pink-noise* (anche noto come 1/f noise, dato che il peso corrisponderebbe al reciproco della frequenza).

1.3. Obiettivi del generatore

Per la realizzazione del generatore di Perlin noise sono stati posti i seguenti obiettivi:

- configurabilità della generazione;
- tempi di esecuzione rapidi;
- output in un formato standard.

Il generatore deve permettere un'ampia configurazione per la generazione di rumore. Tra i parametri di configurazione si è deciso di includere:

- risoluzione: la risoluzione della noise map, di dimensione quadrata;
- densità di rumore: la frequenza iniziale del rumore, perciò la dimensione iniziale del lattice di gradienti;
- numero di ottave: il numero di ottave che si desidera calcolare;
- falloff: il fattore moltiplicato iterativamente al peso delle ottave;
- valore assoluto: se forzare o meno il valore assoluto per le altezze.

Si pone molta importanza sui tempi d'esecuzione. Alcune scelte implementative, tra le quali quelle riguardanti la memorizzazione dei dati, sono state influenzate da questo obiettivo.

Il risultato finale deve essere salvato su memoria di massa con un formato leggibile e facilmente accessibile.

2. Implementazione

Il generatore sviluppato è in grado di produrre una noise map bidimensionale sfruttando la funzione Perlin noise, permettendo una configurazione della generazione di rumore e salvando il risultato finale su un file PNG in una destinazione definibile dall'utente.

In questo capitolo verranno discusse le scelte implementative del progetto del simulatore. Si argomenteranno le scelte inerenti ai parametri di configurazione, all'allocazione di memoria, alla generazione di gradienti e alla generazione del rumore.

2.1. Parametri di configurazione

È possibile configurare la generazione di Perlin noise immettendo in riga di comando i valori dei parametri. I parametri che sono stati implementati sono:

- `OutputPath`: path dell'immagine PNG di output;
- `MapSize`: risoluzione della noise map (di dimensione quadrata);
- `NoiseDensity`: frequenza iniziale del rumore; minore o uguale a `MapSize`;
- `DesiredOctaves`: il numero di ottave che si desidera calcolare;
- `Falloff`: il fattore moltiplicato iterativamente al peso delle ottave;
- `AbsoluteValue`: se forzare o meno il valore assoluto per le altezze.

Ognuno di questi parametri dispone di un valore di default, utilizzato se non ne è stato impostato uno da riga di comando. Per introdurre i parametri all'utente, se nessuno di essi è stato impostato, viene mostrata a schermo una lista che li descrive esaustivamente, indicandone i tipi e i valori di default (Fig. 3).

```
-----------------------------------------------------------------------------------------------
CUDA Perlin Noise Generator; Luigi Rapetta, 2022.
Usage: perlinGenerator OutputPath? MapSize? NoiseDensity? DesiredOctaves? Falloff? AbsoluteValue?
-----------------------------------------------------------------------------------------------
[string]     OutputPath:     path of the output PNG image; default is "./map.png".
[uint]       MapSize:        size of the generated noise image; default is 1024.
[uint]       NoiseDensity:   frequency of the gradients' grid; it's less or equal than MapSize; default is 1.
[uint]       DesiredOctaves: desired number of octaves to compute; default is 1.
[float]      Falloff:        falloff of an octave's weight, incrementally applied at each octave's weight; default is 0.5.
[true/false] AbsoluteValue:  whether to store the absolute values of computed heights; default is false.
```

Fig. 3: lista dei parametri di configurazione, con tipi e valori di default.

In una fase iniziale del programma, i parametri eventualmente impostati vengono controllati per assicurarsi che il loro tipo sia corretto e che rispettino eventuali vincoli (per esempio, `NoiseDensity` deve avere un valore minore o uguale a `MapSize`). Nel caso di errori, saranno utilizzati i valori di default.

Il motivo per il quale `NoiseDensity` deve essere minore o uguale a `MapSize` sta nel fatto che il lattice dei gradienti non può essere più fine della risoluzione della mappa: non si riuscirebbero a definire i quattro gradienti che circonderebbero un pixel P della mappa.

Dopo questa elaborazione iniziale dei parametri, si procede con il calcolo del numero effettivo di ottave da computare. `DesiredOctaves` ne definisce solo un target. Il vincolo discusso sopra potrebbe impedire il calcolo di ulteriori ottave dato che ad ogni ottava si raddoppia la frequenza del rumore, e questo può portare ad utilizzare lattice con una frequenza più alta rispetto alla risoluzione della mappa. Perciò, il numero di ottave effettivamente calcolate è minore o uguale a `DesiredOctaves`.

2.2. Allocazione di memoria

In questo paragrafo si annotano delle scelte legate alle allocazioni di memoria per i gradienti e la noise map.

Il numero di gradienti può essere elevato. Perciò si è deciso di calcolarli direttamente su device, e di conservarli in device memory fino al concludersi dell'applicazione. L'allocazione di gradienti su memoria device cresce esponenzialmente col numero di ottave da calcolare.

Ci sono tre buffer allocati per la noise map: due su device e uno su host. Il primo su device è un buffer di `float`, nel quale vengono salvati i valori delle altezze in un range non definito a priori. Il secondo su device è un buffer di `char`, nel quale vengono salvati i valori finali dei pixel nel range [0, 255]. Sul buffer dell'host, anch'esso in `char`, si riversano da device i valori finali dei pixel. Il buffer su host è quello utilizzato per salvare la mappa in un file PNG.

Si è deciso di impiegare la pinned memory nel tentativo di minimizzare i tempi d'esecuzione. È possibile abilitare il suo impiego grazie a una macro di configurazione. È stato invece deciso di non impiegare la unified memory, in quanto essa rende sensibilmente maggiori i tempi d'esecuzione.

2.3. Esecuzione dei kernel e stream

Si è deciso di impiegare una dimensione fissa per i blocchi di thread. Per i kernel con blocchi 1D è pari a 256. Per i blocchi 2D è pari a 16, per un totale di 256 thread per blocco. Queste dimensioni hanno riportato ottime occupancy durante il profiling effettuato su macchina locale.

I kernel di generazione dei gradienti e del rumore sono eseguiti su stream multipli. Il numero di stream utilizzati per questi kernel è fisso a quattro. È stato scelto l'impiego degli stream solo su questi due kernel in quanto costituiscono la computazione più esigente del programma. Gli altri kernel non gioverebbero particolarmente dagli stream multipli, data la rapidità con la quale vengono eseguiti sullo stream NULL.

Le porzioni di memoria sulle quali operano i kernel lanciati su stream multipli sono partizionate in *slab*. I dati sono, perciò, partizionati in quattro slab. Nel caso di array monodimensionali si possono interpretare come sotto-array contigui, mentre per le matrici bidimensionali si possono visualizzare come partizioni orizzontali. Nel calcolo delle partizioni, si è fatta attenzione a gestire il caso in cui il numero di dati non sia divisibile per quattro: i dati restanti vengono inseriti nella quarta e ultima partizione.

2.4. Generazione e interpretazione dei gradienti

I gradienti sono computati su device con l'impiego della libreria CURAND. I `curandState` vengono generati e conservati direttamente nel kernel, inizializzati con un seed e un numero di sequenza pari all'indice lineare del thread.

Per generare le coordinate (x, y) del gradiente si utilizza una distribuzione uniforme, dalla quale si ricava una coppia di numeri nel range [0.0, 1.0]. Questa coppia viene poi normalizzata nel range [-1.0, 1.0].

Le coordinate vengono salvate sulla memoria device seguendo una logica simile allo *struct di array*: dati N gradienti, i primi N valori nel buffer costituiscono le coordinate x dei gradienti, mentre gli N successivi rappresentano le coordinate y.

Si è deciso di memorizzare i gradienti in questo modo per ottimizzare gli accessi in global memory. Nonostante gli accessi allineati non siano mai garantiti, è possibile godere di un gran numero di accessi contigui.

Gli accessi in memoria saranno sempre contigui durante la memorizzazione dei gradienti. La lettura di questi è contigua nella maggior parte delle volte durante la generazione del rumore.

Per identificare un gradiente sul lattice si fa uso del loro indice lineare (Fig. 4). Esso viene calcolato interpretando il lattice in row-major. Gli indici lineari sono utilizzati nella generazione del rumore per identificare i quattro gradienti attorno a un pixel sulla mappa.

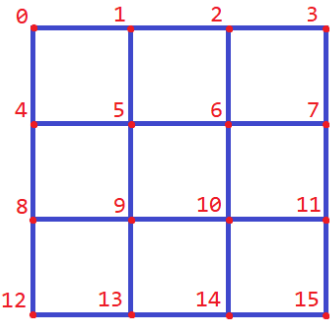

Fig. 4: lattice con indici lineari associati ai gradienti.

Questa modalità di interpretazione del lattice impedisce di garantire che le letture dei gradienti siano sempre contigue. Durante la generazione del rumore, ad ogni thread corrisponde un pixel sulla mappa e, in alcuni casi, un warp di thread può ricoprire più celle del lattice (Fig. 5, a destra). In tale situazione, i thread del warp non richiedono tutti gli stessi gradienti.

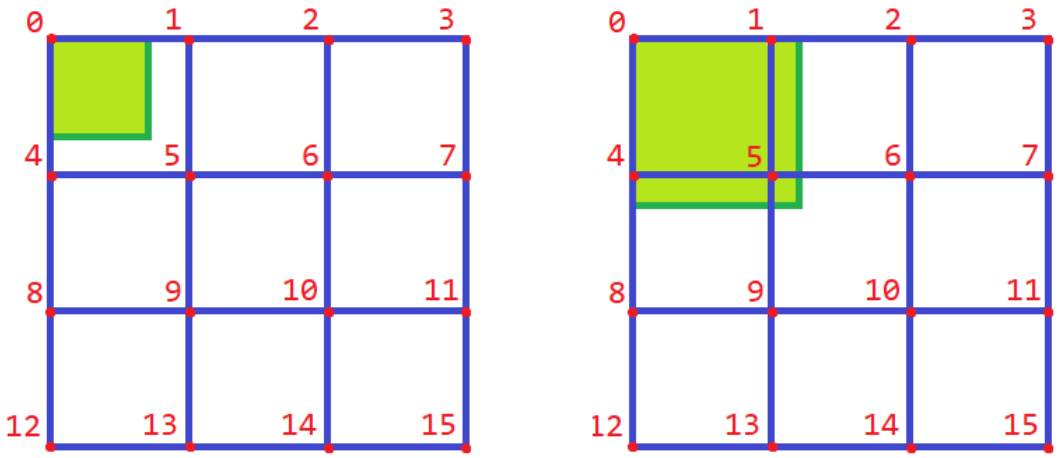

Fig. 5: un warp (verde) potrebbe ricoprire più celle del lattice (caso a destra); in questi casi, la lettura in global memory dei gradienti può non essere ottimale.

Per esempio, prendendo in esame il caso a destra in Fig. 5, se si decidesse di richiedere il gradiente in basso a destra per ogni pixel, il warp (in verde) richiederebbe i gradienti 5, 6, 9 e 10. Chiaramente, ciò può portare a letture non ottimizzate, data la mancanza di contiguità nell'accesso. Nel caso a sinistra in Fig. 5, invece, se si richiedesse il gradiente in basso a destra per ogni pixel, tutti i thread del warp richiederebbero solo il gradiente 5.

La generazione dei gradienti sfrutta, come anticipato precedentemente, gli stream multipli. Le porzioni di memoria sulle quali si memorizzano i gradienti sono partizionate in slab. Ogni stream è dedicato a calcolare un numero di gradienti pari alla dimensione dello slab.

2.5. Generazione del rumore

Durante la generazione del rumore, ad ogni thread corrisponde un pixel sulla mappa. Il kernel `perlinNoise()` calcola il valore di altezza per ogni pixel. Il thread associato a un pixel computerà tutte le ottave di Perlin noise all'interno di un ciclo.

Data la natura additiva nel rumore frattale, un'altezza di un pixel rientra in un intervallo non definito. Perciò, dopo la generazione del rumore, è necessario calcolare un intervallo [min, max] nel quale rientrano tutte le altezze della mappa e convertirle da quest'ultimo all'intervallo [0, 255]. Il processo di normalizzazione avviene in un due kernel successivi a `perlinNoise()`, discussi nel prossimo paragrafo.

Nel ciclo dedicato a calcolare l'altezza di un pixel per un'ottava, la prima operazione ad essere eseguita è l'identificazione e la lettura dei quattro gradienti che circondano il pixel associato al thread. I gradienti vengono letti, come descritto nel paragrafo precedente, tramite il loro indice lineare.

Dopodiché si passa al calcolo delle quattro altezze relative ai quattro gradienti selezionati, e alla loro interpolazione. Quest'ultima è l'altezza associata al pixel dell'ottava che il thread sta calcolando. Se è richiesto dalla configurazione, si sostituisce l'altezza con il suo valore assoluto.

L'altezza calcolata per questa ottava viene moltiplicato con il fattore di falloff e sommata al valore dell'altezza finale. Sostanzialmente, si aggiunge all'altezza finale il contributo dato dall'ottava appena calcolata.

Alla fine del ciclo, dopo che tutte le ottave sono state computate per un determinato pixel, si ha finalmente a disposizione l'altezza finale. Quest'ultima, non ancora normalizzata, viene memorizzata sulla matrice passata come argomento della funzione.

Anche questa fase dell'applicazione fa uso degli stream multipli e, anche in questo caso, le porzioni di memoria sulle quali si memorizzano le altezze non normalizzate sono partizionate in slab.

2.6. Normalizzazione e fasi finali

Dopo la generazione del rumore è richiesta la normalizzazione delle altezze da un range arbitrario all'intervallo [0, 255], il quale può essere interpretato come il canale del colore di un'immagine monocromatica.

In preparazione alla normalizzazione, si definisce l'intervallo di valori nel quale ricadono le altezze calcolate nella fase precedente. In altre parole, si devono identificare un minimo e un massimo tra queste altezze, per poi usarli come estremi dell'intervallo [min, max].

Per definire [min, max] si fa uso del kernel `computeHeightsRange()` il quale accetta, come argomenti rilevanti, la mappa non normalizzata e un buffer per memorizzare l'intervallo. Il kernel associa un thread a un pixel sulla mappa.

Un thread preleva dalla mappa l'altezza relativa al proprio pixel. Le altezze lette da un blocco di thread vengono memorizzate in due array di memoria shared, `blockMin` e `blockMax`. Dopo una dovuta sincronizzazione, si esegue la parallel reduction con warp unrolling su entrambi gli array per ricavare il minimo e il massimo da questi.

È stata dedicata particolare attenzione a gestire il caso in cui un thread di un blocco non sia mappato su un pixel. In tal caso, il thread inizializza il proprio slot in `blockMin` e `blockMax` con un valore di defalut che non compromette l'accuratezza del calcolo del minimo e del massimo locale al blocco. Di seguito si presenta l'estratto di codice che gestisce la situazione appena descritta:

```
[...]
// Check sanity.
int linearIdx = blockIdx.x * blockDim.x + threadIdx.x;
if (linearIdx >= mapSize * mapSize) {
    // Fill shared buffers with dummy values; then return.
    blockMin[threadIdx.x] = FLT_MAX;
    blockMax[threadIdx.x] = -FLT_MAX;
    return;
}
[...]
```

Nonostante crei una minima divergenza, ciò è necessario per rendere la reduction operabile, dato che quest'ultima richiede una dimensione dei dati pari alla dimensione di blocco.

Dopo aver trovato i minimi e i massimi locali ai blocchi, si eseguono delle operazioni atomiche su float per memorizzare l'altezza massima e minima dell'intera griglia su memoria globale. Il buffer per memorizzare l'intervallo adesso conserva gli estremi del range [min, max]. Di seguito si riporta un estratto del codice dedicato a memorizzare gli estremi a livello di griglia:

```
[...]
// Perform atomic operations to get the grid min and max.
if (threadIdx.x == 0) atomicMinFloat(&heightRange[0], blockMin[0]);
if (threadIdx.x == 0) atomicMaxFloat(&heightRange[1], blockMax[0]);
[...]
```

Il valore di un estremo a livello di griglia (nel buffer `heightRange`) viene confrontato atomicamente con quello ricavato da un blocco (`blockMin[0]` per il minimo e `blockMax[0]` per il massimo).

Si è deciso di usufruire della shared memory durante la parallel reduction per i seguenti motivi:

- ridurre drasticamente gli accessi in memoria globale;
- evitare di modificare le altezze della mappa sulla matrice data in input.

Successivamente alla definizione dell'intervallo, si passa alla vera normalizzazione lanciando il kernel `normalizeMap()`. Gli vengono passati come argomenti la mappa da normalizzare, il buffer per memorizzare i valori normalizzati, il range [min, max] e la risoluzione della mappa. Il kernel associa un thread a un pixel sulla mappa.

Un thread preleva dalla mappa l'altezza relativa al proprio pixel. L'altezza viene convertita da [min, max] a [0, 255] e memorizzata come char nel buffer di output. Terminata l'esecuzione del kernel, si ha finalmente a disposizione la matrice di valori finali scrivibili su file PNG.

Le altezze normalizzate vengono riversate nella porzione di memoria host utilizzata dalla routine di scrittura su file PNG. Quest'ultima fa parte di una libreria di terze parti chiamata `stb_image_write.h`{2}.

L'intera fase di normalizzazione non fa uso di più stream. I tempi di esecuzione relativamente ridotti di questi due kernel[2], verificati durante il profiling su macchina locale, non giustificano il loro impiego.

2 Si è verificato che ottimizzazioni sui processi di normalizzazione non migliorano significativamente i tempi d'esecuzione.

3. Test

Di seguito si mostrano i risultati di test svolti con diversi parametri di configurazione. Si è deciso di minimizzare la risoluzione delle immagini qui mostrate per evitare l'appesantimento del documento. Il capitolo dedicato al profiling discuterà risultati ottenuti con risoluzioni della mappa molto elevate. È possibile reperire le seguenti immagini tra gli allegati al report.

3.1. Frequenza del rumore

Si mostrano esempi di mappe con diverse frequenze del rumore.

Fig. 6: `NoiseDensity` *= 2;* `DesiredOctaves` *= 1;* `Falloff` *= 0.5;* `AbsoluteValue` *= false*

Fig. 7: `NoiseDensity` *= 4;* `DesiredOctaves` *= 1;* `Falloff` *= 0.5;* `AbsoluteValue` *= false*

Fig. 8: `NoiseDensity` *= 8;* `DesiredOctaves` *= 1;* `Falloff` *= 0.5;* `AbsoluteValue` *= false*

3.2. Numero di ottave

Si mostrano esempi di mappe con diversi numeri di ottave da computare.

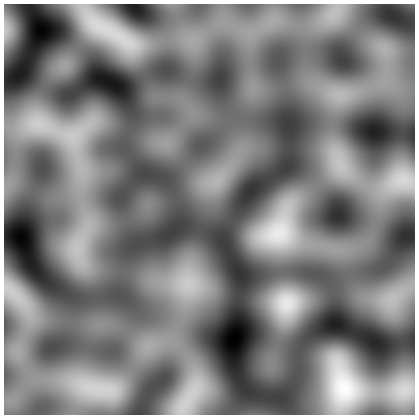

Fig. 9: `NoiseDensity` *= 8;* `DesiredOctaves` *= 1;* `Falloff` *= 0.5;* `AbsoluteValue` *= false*

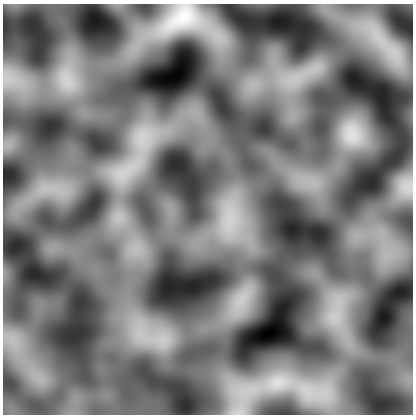

Fig. 10: `NoiseDensity` *= 8;* `DesiredOctaves` *= 1;* `Falloff` *= 0.5;* `AbsoluteValue` *= false*

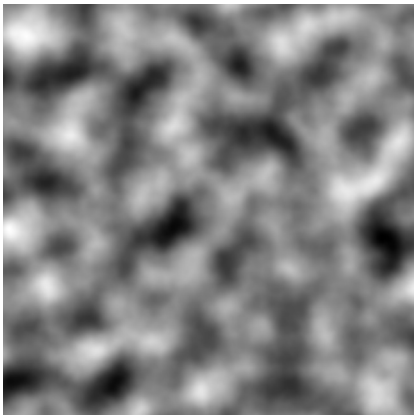

Fig. 11: `NoiseDensity` *= 8;* `DesiredOctaves` *= 4;* `Falloff` *= 0.5;* `AbsoluteValue` *= false*

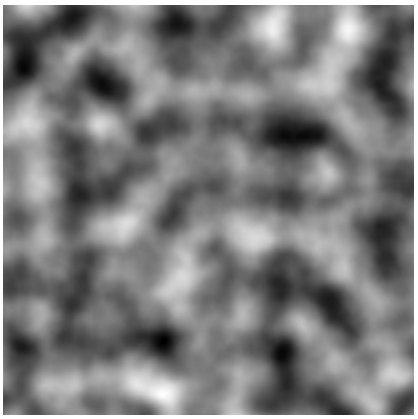

Fig. 12: `NoiseDensity` *= 8;* `DesiredOctaves` *= 8;* `Falloff` *= 0.5;* `AbsoluteValue` *= false*

3.3. Falloff

Si mostrano esempi di mappe con diversi valori di falloff.

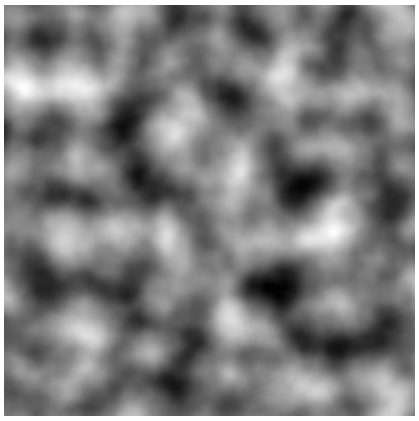

Fig. 13: `NoiseDensity` *= 8;* `DesiredOctaves` *= 8;* `Falloff` *= 0.5;* `AbsoluteValue` *= false*

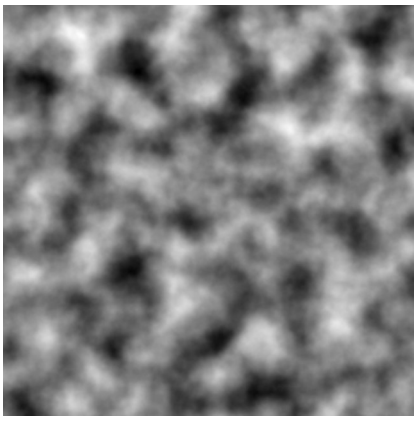

Fig. 14: `NoiseDensity` *= 8;* `DesiredOctaves` *= 8;* `Falloff` *= 0.75;* `AbsoluteValue` *= false*

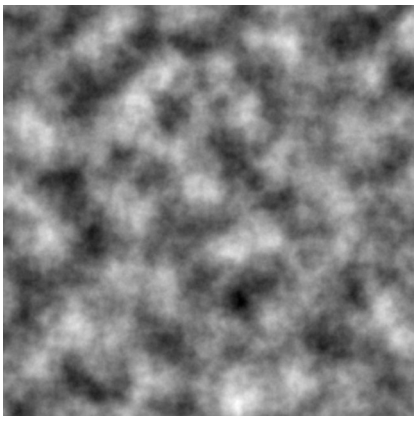

Fig. 15: `NoiseDensity` *= 8;* `DesiredOctaves` *= 8;* `Falloff` *= 1.0;* `AbsoluteValue` *= false*

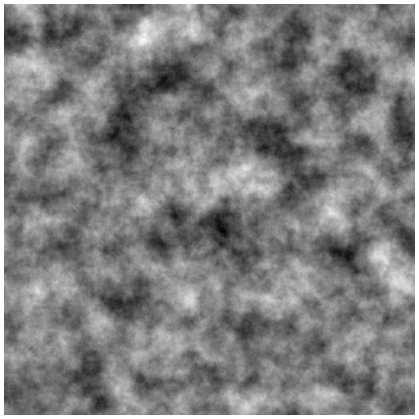

Fig. 16: `NoiseDensity` *= 8;* `DesiredOctaves` *= 8;* `Falloff` *= 1.25;* `AbsoluteValue` *= false*

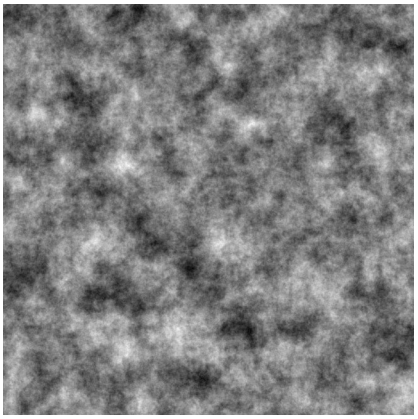

Fig. 17: `NoiseDensity` *= 8;* `DesiredOctaves` *= 8;* `Falloff` *= 1.5;* `AbsoluteValue` *= false*

3.4. Valore assoluto

Si mostrano esempi di mappe con diversi falloff e il calcolo del valore assoluto delle ottave attivo.

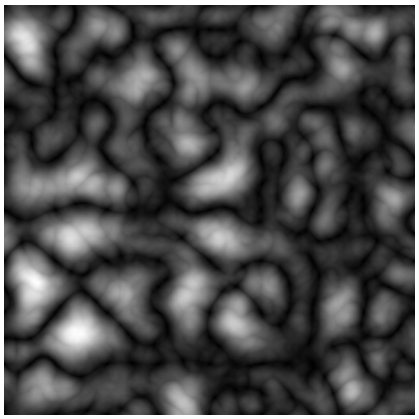

Fig. 18: `NoiseDensity` *= 8;* `DesiredOctaves` *= 8;* `Falloff` *= 0.5;* `AbsoluteValue` *= true*

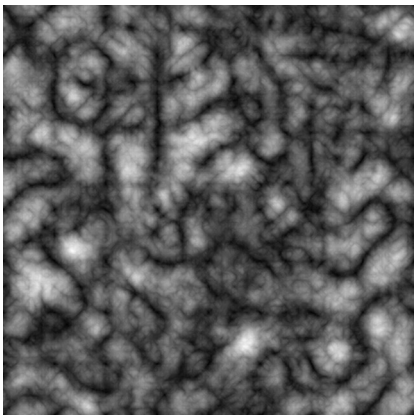

Fig. 19: `NoiseDensity` *= 8;* `DesiredOctaves` *= 8;* `Falloff` *= 1.0;* `AbsoluteValue` *= true*

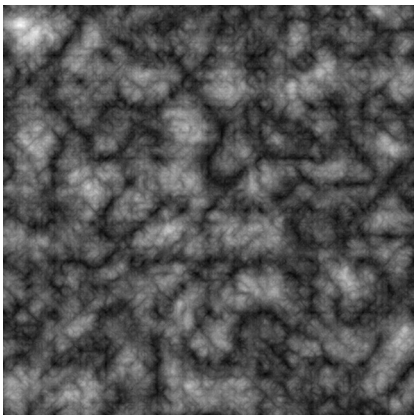

Fig. 20: `NoiseDensity` *= 8;* `DesiredOctaves` *= 8;* `Falloff` *= 1.25;* `AbsoluteValue` *= true*

4. Profiling

Nel capitolo corrente vengono discussi i risultati della fase di profiling, esponendo i risultati ed eventualmente argomentando problemi riscontrati.

Durante il profiling sono state utilizzate una macchina locale e una remota.

Le specifiche della macchina locale:

- GPU: Nvidia RTX 2070 SUPER 8GB
- CPU: AMD Ryzen 7 3700X 8-Core Processor @ 3.60 GHz

La macchina remota è messa a disposizione dal servizio Google Colaboratory. Le specifiche della macchina remota:

- GPU: Nvidia Tesla T4 16GB
- CPU: Intel Xeon CPU @ 2.00 GHz

Per il profiling su macchina locale e remota è stato usato Nsight. Su CLI, lo strumento è stato lanciato con il seguente comando e la seguente configurazione:

```
nv-nsight-cu-cli ./PerlinGenerator.exe ./map.png 16384 8 8 1 false
```

È stato anche utilizzato Nsight Systems per il profiling su macchina locale. Esso offre una GUI che permette di visualizzare i kernel eseguiti sula GPU, distinguendo sugli stream. Questo tool è stato usato per verificare l'esecuzione degli stream multipli.

Per l'analisi delle metriche non è stato utilizzato nvprof, in quanto non più supportato dalla Compute Capability 7.5. È stato solamente utilizzato per analizzare i tempi d'esecuzione su macchina remota ed avere un risultato rapidamente leggibile sul presente documento.

I risultati del profiling sono reperibili tra gli allegati del presente documento.

Nota importante: lo sviluppo dell'applicazione è stato principalmente svolto su macchina locale; l'impiego della macchina remota è puramente per scopo dimostrativo e di confronto.

4.1. Tempi di esecuzione e speedup

Si riportano i tempi d’esecuzione e gli speedup sulla macchina locale:

Configurazione	Tempo GPU	Tempo CPU	Speedup
MapSize = 4096 NoiseDensity = 8 DesiredOctaves = 8 Falloff = 1.0 AbsoluteValue = false	952.053345ms	3236.917969ms	3.399933
MapSize = 8192 NoiseDensity = 8 DesiredOctaves = 8 Falloff = 1.0 AbsoluteValue = false	2853.263672ms	11691.353516ms	4.097537
MapSize = 16384 NoiseDensity = 8 DesiredOctaves = 8 Falloff = 1.0 AbsoluteValue = false	7475.445312ms	43766.203125ms	5.854661
MapSize = 32768 NoiseDensity = 8 DesiredOctaves = 8 Falloff = 1.0 AbsoluteValue = false	24681.945312ms	169959.218750ms	6.885973

Si riportano i tempi d’esecuzione e gli speedup sulla macchina remota:

Configurazione	Tempo GPU	Tempo CPU	Speedup
MapSize = 4096 NoiseDensity = 8 DesiredOctaves = 8 Falloff = 1.0 AbsoluteValue = false	12217.083984ms	13393.814453ms	1.096318
MapSize = 8192 NoiseDensity = 8 DesiredOctaves = 8 Falloff = 1.0 AbsoluteValue = false	19251.974609ms	48262.578125ms	2.506890
MapSize = 16384 NoiseDensity = 8 DesiredOctaves = 8 Falloff = 1.0 AbsoluteValue = false	43348.753906ms	185089.671875ms	4.269781
MapSize = 32768 NoiseDensity = 8 DesiredOctaves = 8 Falloff = 1.0 AbsoluteValue = false	126760.085938ms	717224.125000ms	5.658123

Durante il profiling, si è notato come il calcolo più esigente dipenda anche dalla GPU sulla quale si esegue il programma. Sulla macchina remota, la generazione dei gradienti impiega molto più tempo rispetto alla generazione del rumore, mentre vale il contrario sulla macchina locale.

Qui si riporta un estratto dal profiling su macchina locale; notare la minore durata dell'intera generazione dei gradienti rispetto a una singola chiamata di `perlinNoise()`:

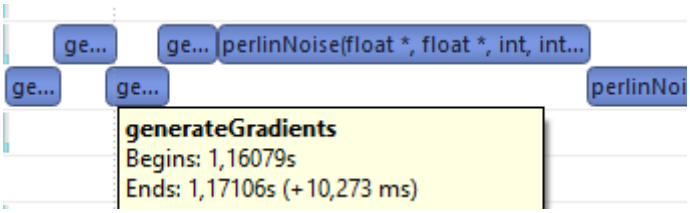

Fig. 21: eseguendo il programma in locale, i kernel per la generazione dei gradienti durano circa 10ms l'uno, mentre quelli per la generazione del rumore impiegano circa 60ms ciascuno.

Qui si riporta un estratto dell'output di nvprof su macchina remota:

```
            Type  Time(%)      Time     Name
GPU activities:   96.75%  9.98958s     generateGradients(…)
                   2.85%  294.32ms     perlinNoise(…)
                   0.20%  20.470ms […] [CUDA memcpy DtoH]
                   0.09%  9.5061ms     computeHeightRange(…)
                   0.06%  6.1530ms     normalizeMap(…)
                   0.04%  4.6211ms     [CUDA memset]
```

Sulla macchina remota, `generateGradients()` risulta essere un collo di bottiglia per le prestazioni molto importante. Nel cercare di mitigare il problema, si è provato a migrare fuori dal kernel la conservazione dello stato CURAND, allocando da host un buffer di memoria device di tipo `curandState_t` e passandolo come argomento al kernel. Ciò è stato vano, in quanto non ha minimamente portato a miglioramenti nelle performance.

4.2. Occupancy

L'occupancy riscontrata risulta molto buona. Per il kernel `generateGradients()`, l'occupancy rientra tra il 94% e il 95% su macchina locale, mentre cade tra il 96% e il 97% su macchina remota. Per entrambe le macchine, l'occupancy di `perlinNoise()` si registra attorno il 97%, quella di `computeHeightsRange()` si aggira sul 60% e quella di `normalizeMap()` oscilla attorno l'80%. Si possono reperire i risultati completi tra gli allegati.

4.3. Stream multipli

Come discusso in precedenza, il programma sfrutta l'esecuzione su quattro stream distinti. Le fasi di generazione dei gradienti e del rumore fanno uso di questi stream.

Nonostante i kernel vengano eseguiti sui propri stream, durante il profiling è stato riscontrato che essi non sono eseguiti in parallelo (Fig. 22).

Qui sotto viene riportata la linea temporale dell'esecuzione su macchina locale dei kernel sui quattro stream:

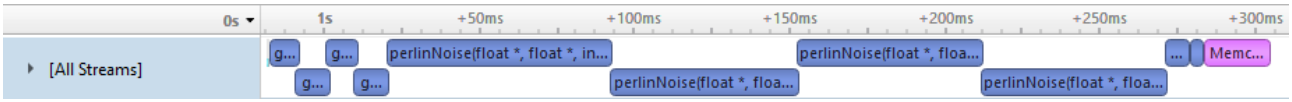

Fig. 22: esecuzione in locale con stream multipli; notare il problema di non parallelizzazione dei kernel.

Il problema è verificabile sulla macchina remota. Qui si riporta un estratto dell'output di nvprof:

```
    Time(%)      Time  Calls       Avg     Name
     96.75%  9.98958s      4  2.49740s     generateGradients(…)
      2.85%  294.32ms      4  73.579ms     perlinNoise(…)
[…]   0.20%  20.470ms      1  20.470ms […] [CUDA memcpy DtoH]
      0.09%  9.5061ms      1  9.5061ms     computeHeightRange(…)
      0.06%  6.1530ms      1  6.1530ms     normalizeMap(…)
      0.04%  4.6211ms      3  1.5403ms     [CUDA memset]
```

Prendendo in esame l'output precedente, anche se una sola chiamata di `perlinNoise()` impiega mediamente 74ms, il tempo necessario per eseguire tutte e quattro le chiamate è circa 295ms; l'esecuzione non è parallela.

Dopo una lunga fase di ricerca per una soluzione al problema, sono state identificate due possibili cause:

- i tempi d'esecuzione dei kernel sono talmente piccoli da non permettere allo scheduler di farli eseguire in parallelo;
- un singolo kernel occupa talmente tante risorse da non permettere l'esecuzione parallela di più kernel.

Purtroppo, entrambe le situazioni sembrano essere osservabili su entrambe le macchine. Infatti, i tempi d'esecuzione, a parte per `generateGradients()` su macchina remota, non superano mai la soglia del secondo. Inoltre, il profiling espone l'utilizzo elevato delle risorse da parte di `perlinNoise()` e `generateGradients()`: i valori di `Compute (SM) [%]` e `Memory [%]` superano la soglia del 90% su entrambe le macchine.

Su entrambi gli output di Nsight, compare di frequente la seguente informazione:

```
The kernel is utilizing greater than 80.0% of the available compute or memory
performance of the device. To further improve performance, work will likely need
to be shifted from the most utilized to another unit. Start by analyzing
workloads in the Compute Workload Analysis section.
```

È stato tentato empiricamente di abbassare il numero di stream a 2, nel cercare di rendere più duratura l'esecuzione di un singolo kernel e dare la possibilità allo scheduler di eseguirli in parallelo, ma ciò non ha portato a risultati diversi.

Al momento della scrittura del documento, non sono state trovate soluzioni per abbassare il livello di risorse computazionali occupate durante l'esecuzione di un singolo kernel. Non è chiaro se una buona soluzione implicherebbe un completo ripensamento del funzionamento del programma o l'impiego di un paradigma multi GPU.

Nonostante la situazione non rosea, si è deciso di mantenere l'uso di stream multipli per supportare l'eventuale esecuzione del programma su GPU high-end e con moli di dati enormi.

5. Bibliografia

[1] I. Millington, *AI for Games* (3rd ed.), CRC Press, 2020

6. Sitografia

{1} *Wikipedia*, *Perlin noise*, https://en.wikipedia.org/wiki/Perlin_noise
{2} Sean Barrett, *stb*, https://github.com/nothings/stb