SIMULAZIONE DI UN FLUIDO REAL-TIME

Un progetto per il corso di GPU Computing, A.A 2021/2022

Di Gianluca Iacchini, Mat. 942386

Introduzione

Questa relazione documenta l'implementazione di una simulazione di un fluido in tempo reale per il corso di GPU Computing dell'Università statale di Milano.
Lo scopo del progetto è quello di dimostrare le potenzialità di una applicazione GPGPU rispetto alla classica implementazione sequenziale; per questo motivo il progetto è separato in due moduli, uno che implementa la simulazione interamente sulla CPU, ed uno che implementa la simulazione sfruttando anche la GPU tramite CUDA.

Librerie e tools utilizzati

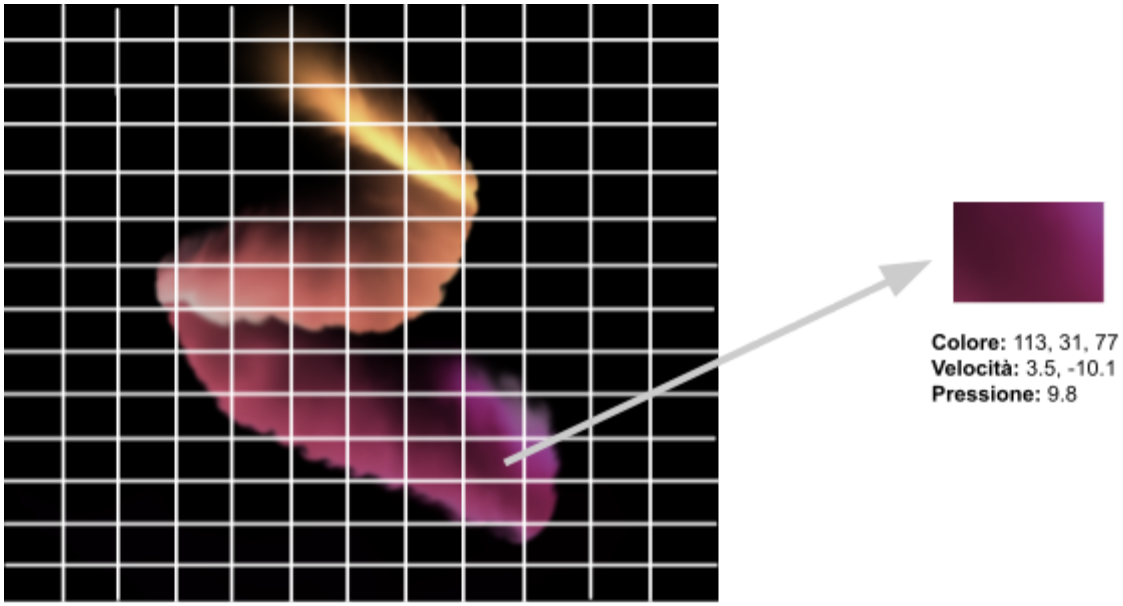
La simulazione è stata scritta in C++ e CUDA. Le uniche librerie usate oltre quelle standard sono GLFW e GLAD; queste permettono di utilizzare OpenGL per poter visualizzare la simulazione.

Il progetto è stato sviluppato utilizzando Visual Studio 2022, mentre per effettuare il profiling dell'applicazione è stato utilizzato Nsight Compute.

Simulazione

La simulazione di un fluido può essere implementata tramite due metodi: il metodo Lagrangiano e quello Euleriano.
La tecnica Lagrangiana usa delle particelle per tenere traccia del movimento del fluido. Ogni punto del fluido può essere pensato come una singola particella con le sue particolari proprietà, come velocità, temperatura o colore.
La tecnica Euleriana invece monitora dei punti fissi nello spazio del dominio del fluido e tiene traccia di come le quantità variano in ognuno di questi punti; di solito questa tecnica viene implementata utilizzando una griglia.

In questo progetto è stata usata la tecnica Euleriana, questo perché le GPU sono ideali per lavorare con strutture come le griglie, in questo modo possiamo sfruttare al massimo l'alto grado di parallelismo presente nel device.



Il dominio del fluido è diviso in celle della stessa dimensione, ognuna delle quali contiene delle informazioni riguardo al fluido in quel punto.

Equazioni di Navier-Stokes

Un fluido può essere descritto utilizzando le equazioni di Navier-Stokes. Queste sono un sistema di equazioni alle derivate parziali e vengono di solito scritte nella seguente forma:

$$\frac{\delta \vec{u}}{\delta u} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho}\nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u}$$

$$\nabla \cdot \vec{u} = 0$$

Dove u rappresenta la velocità, p la pressione, ν la viscosita e g la forza di gravità. Queste possono essere riscritte nella seguente maniera

$$\frac{\delta \vec{u}}{\delta u} = -(\vec{u} \cdot \nabla \vec{u}) - \frac{1}{\rho}\nabla p + \nu \nabla^2 \vec{u} + F$$

$$\nabla \cdot \vec{u} = 0$$

Dividiamo le equazioni in più parti al per comprenderle meglio.

La prima parte $-(\vec{u} \cdot \nabla \vec{u})$ è chiamata **avvezione.** L'avvezione è il fenomeno per cui la velocità del fluido causa lo spostamento di altre quantità (come temperatura o del colorante) lungo il suo flusso. La velocità stessa è influenzata dall'avvezione del fluido; questa proprietà si chiama auto-avvezione.

La seconda parte, $-\frac{1}{\rho}\nabla p$, fa entrare in gioco la **pressione**. Quando una forza viene applicata al fluido non si propaga istantaneamente ma “costringe” le molecole a spingersi a vicenda.

La parte finale $\nu\nabla^2\vec{u} + F$ tiene conto di **forze esterne**, come la gravità. Queste possono essere applicate a delle regioni specifiche del fluido o all’intero volume.

La seconda equazione $\nabla \cdot \vec{u} = 0$ pone una restrizione importante, il fluido che consideriamo deve essere incompressibile, ossia il suo volume non deve variare con il passare del tempo.

Per ricapitolare, quello che le equazioni dicono è che dato un fluido **incompressibile**, la sua velocità può essere calcolata in diversi step: **avvezione**, **pressione** e **forze esterne**. Una volta calcolata la velocità possiamo usarla per capire come le altre proprietà del fluido si spostano all’interno di esso (usando sempre lo step dell’**avvezione**).

Algoritmo

L’algoritmo per la simulazione di un fluido è composto da una serie di passi che vengono ripetuti ad ogni frame.

```
void compute_one_simulaton_frame()
{
    u = advect(u);

    // Advect other quantities here

    u = addForces(u); // Vorticity, Diffuse, Gravity, Impulse,  etc.

    p = computePressure(u);
    u = subtractPressureGradient(u, p);
}
```

La simulazione è divisa in una serie di step, i quali hanno il compito di calcolare i valori di ogni cella. Ad ogni time step t_i della simulazione, dei nuovi valori per ogni cella vengono calcolati basandosi sui valori presenti al time step t_{i-1} dentro di esse e dentro le celle a loro vicine.

Il valore più importante della simulazione è la velocità. La maggior parte della simulazione infatti si occupa di ricalcolare i nuovi valori della velocità in ogni cella, questi verranno poi usati nel time step successivo per calcolare i valori delle altre proprietà del fluido.

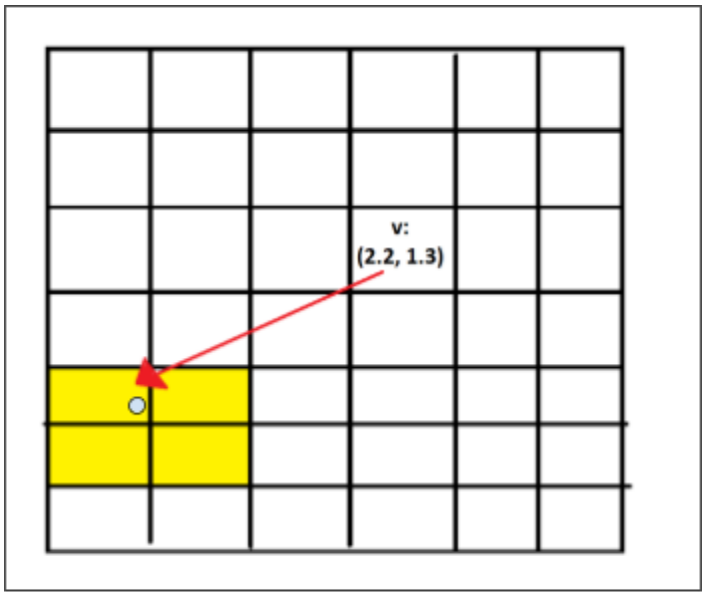
Step della simulazione

Di seguito vengono presentati i vari step della simulazione con una breve descrizione. Non entriamo troppo nei dettagli fisici della simulazione in quanto essi vanno oltre lo scopo del progetto, il cui obiettivo è quello di dimostrare l'efficienza di una applicazione GPGPU.

Avvezione

Come abbiamo appena visto, l'avvezione è il fenomeno secondo il quale la velocità del fluido trasporta se stessa e altre quantità lungo il flusso.
Per calcolarla immaginiamo di avere una particella in ogni punto della griglia. Vogliamo sapere in quale cella va a finire dopo un certo lasso di tempo δt; una volta trovata la destinazione aggiorniamo i valori della cella di arrivo con le proprietà della cella di partenza.
Vi è in realtà un grosso problema nell'utilizzare questo approccio: per valori troppo alti di δt la simulazione potrebbe diventare numericamente instabile.
Pertanto quello che facciamo è effettuare l'operazione inversa: data una cella qualsiasi ci chiediamo da quale cella proviene la particella immaginaria dentro di essa. Dopodiché aggiorniamo il valore della cella corrente con quelli della cella di partenza.
È dimostrabile che con questa operazione inversa la simulazione rimane numericamente stabile, indipendentemente dai valori di δt.

Vi è un ultimo accorgimento da fare, poiché lavoriamo in uno spazio discreto (una griglia), difficilmente la particella immaginaria avrà come punto di partenza il centro esatto di un'altra cella, ma è invece più probabile che sia spostata di un qualche offset. Pertanto dopo aver rintracciato il punto di partenza effettuiamo una interpolazione bilineare tra le quattro celle più vicine, il risultato di tale operazione sarà il nuovo valore per la nostra cella.



Abbiamo rintracciato il punto di partenza di una particella, ma non cade nel centro di una cella. Pertanto effettuiamo una interpolazione bilineare.

Pressione
Per poter calcolare la pressione, usiamo una soluzione iterativa chiamata *Jacobi Iteration.* Questa è un'equazione che viene ripetuta un numero limitato di volte, utilizzando l'output dell'iterazione precedente come input per quella successiva. Un maggior numero di iterazioni significa un risultato più preciso ma allo stesso tempo comporta anche delle performance peggiori.

Diffusione
La diffusione regola come il fluido si diffonde in base alla sua viscosità. La diffusione viene calcolata esattamente come la pressione, infatti l'algoritmo è lo stesso; quello che cambia sono i valori dei vari parametri (coefficienti delle formule, vettori di input, numero di iterazioni).

Vorticità
La vorticità regola quanto è propenso il fluido a creare dei vortici e degli arricciamenti. La vorticità è un tipo di forza esterna.

Implementazione

Entriamo ora nei dettagli dell'implementazione del progetto. Come abbiamo visto nelle sezioni precedenti, la simulazione ha il compito di calcolare ad ogni step un nuovo valore per la velocità, la quale a sua volta viene usata per calcolare dei nuovi valori per le altre proprietà.

Vi è quindi una parte sequenziale dell'algoritmo, dato dal fatto che ogni step utilizza come input il valore della velocità calcolato in output dallo step precedente, ed anche una parte fortemente parallela data dal fatto che gli step devono calcolare dei nuovi valori per ogni cella della griglia.

Implementazione sequenziale

Partiamo dall'implementazione sequenziale dell'algoritmo.

```
/* Computes a single time step of the simulation. dt: Time passed since
last frame *\
void on_time_step(float dt)
{
    //Advect
    advect_velocity(dt, aDecay);
    advect_color(dt, aDecay);

    //Vorticity
```

```
    vorticity_confinement(dt);

    // Diffuse
    diffuse(dt, density_diffusion, velocity_diffusion);

    // Force
    apply_color_and_force(dt);

    //Pressure
    pressure_iteration();

    //Gradient subtraction
    project_pressure();
}
```

Per ogni proprietà del fluido creiamo due array di dimensione GRID_WIDTH * GRID_HEIGHT. Le proprietà del fluido del quale teniamo traccia sono la velocità, la pressione e il colorante (che serve per visualizzare la simulazione).

Il motivo per cui abbiamo bisogno di due array per ogni proprietà e dato dal fatto che ogni volta che calcoliamo i nuovi valori per ogni cella, utilizziamo le informazioni presenti in alcune delle celle adiacenti. Se non usassimo un secondo array allora alcune celle utilizzerebbero i valori già aggiornati presenti nelle celle vicine anziché utilizzare quelli presenti nel passo temporale precedente.

Gli step sono diversi tra loro, ma seguono tutti uno schema generale (con eccezione dell'avvezione di cui abbiamo già visto il funzionamento):

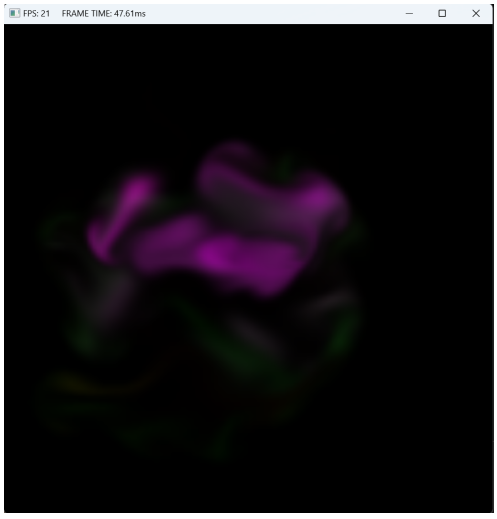
```
Foreach Cell c_{x,y} in Old_Grid:
    c_{x-1,y} = leftCell(c);
    c_{x+1,y} = rightCell(c);
    c_{x,y-1} = bottomCell(c);
    c_{x,y+1} = topCell(c);
    New_Grid[x,y] = compute_new_value(c_{x,y}, c_{x-1,y}, c_{x+1,y}, c_{x,y-1}, c_{x,y+1}, dt);
```

Una volta che abbiamo calcolato la simulazione per un intero time step possiamo visualizzarla a schermo, per farlo ci appoggiamo su OpenGL.

All'inizio dell'applicazione creiamo una texture con dimensione pari a quella della griglia, ogni volta che completiamo un frame coloriamo questa texture con i valori presenti nell'array del colorante, dopodiché la mostriamo a schermo.

```
/* compute the simulation and display it */
void on_frame()
{
    float dt = computeTimestep();
    // Compute simulation physics
    on_time_step(dt);

    // Set color dye data to OpenGL input format
    FOR_EACH_CELL{
        float R = old_color[y * GRID_WIDTH + x].x;
        float G = old_color[y * GRID_WIDTH + x].y;
        float B = old_color[y * GRID_WIDTH + x].z;

        pixels[y * GRID_WIDTH + x] = make_uchar4(fminf(255.0f, 255.0f * R),
fminf(255.0f, 255.0f * G), fminf(255.0f, 255.0f * B), 255);
    }

    // upload pixels to texture
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, GRID_WIDTH, GRID_HEIGHT, 0,
GL_RGBA, GL_UNSIGNED_BYTE, pixels);

    // display texture to screen
    drawTexture();
}
```

Di seguito il risultato della simulazione su CPU



Implementazione GPGPU

Vediamo ora l'implementazione che sfrutta anche la GPU. Come abbiamo visto l'algoritmo è composto da una parte sequenziale, data dal fatto che dobbiamo aspettare i risultati dello step precedente per eseguire quello successivo, che non beneficia molto dalla potenza extra che ci fornisce il device; tuttavia il vero vantaggio risiede nel fatto che ora possiamo calcolare i valori di ogni cella in parallelo, riducendo di molto i tempi di esecuzione del programma.

```
void on_time_step(float dt)
{
	// advect
	advect << <numBlocks, threadsPerBlock >> > (velocityField, dt);
	advect << <numBlocks, threadsPerBlock >> > (dyeColorField,
velocityField, dt);

	// curls and vorticity
	computeVorticity <<<numBlocks, threadsPerBlock >> > (velocityField,
vorticityField, dt);

	// diffuse velocity and color
	diffuseVel << <numBlocks, threadsPerBlock, 0, stream_0 >> >
(velocityField, dt);
	diffuseCol << <numBlocks, threadsPerBlock, 0, stream_1 >> >
(dyeColorField, dt);


	// apply force
	CUDA_CALL(cudaStreamSynchronize(stream_0));
	CUDA_CALL(cudaStreamSynchronize(stream_1));
	applyForce << <numBlocks, threadsPerBlock >> > (velocityField,
dyeColorField, dt);

	// compute pressure
	computeDivergence << <numBlocks, threadsPerBlock >> >
(divergenceField, velocityField);
	computePressureImpl << <numBlocks, threadsPerBlock >> >
(divergenceField, pressureField, dt);

	// Gradient subtraction
	project <<<numBlocks, threadsPerBlock >> > (velocityField,
pressureField);
}
```

Una volta calcolata la simulazione per un intero time step, possiamo iniziare a popolare la texture con i valori del colorante. A differenza della simulazione su CPU dove questo comportava un trasferimento di dati da host a device, nella simulazione tramite CUDA i dati non passano mai per l'host grazie a delle funzioni di interoperabilità tra CUDA e OpenGL.

```
Void on_frame()
{
    float dt = computeTimestep();

    on_time_step(dt);

    // Convert data to OpenGL input format
    convertToOpenGLInput<<<numBlocks, threadsPerBlock >> >
(textureColorField, dyeColorField);

    // Write directly to texture from device
    writeToTexture << <numBlocks, threadsPerBlock >> > (textureObj,
textureColorField);
}
```

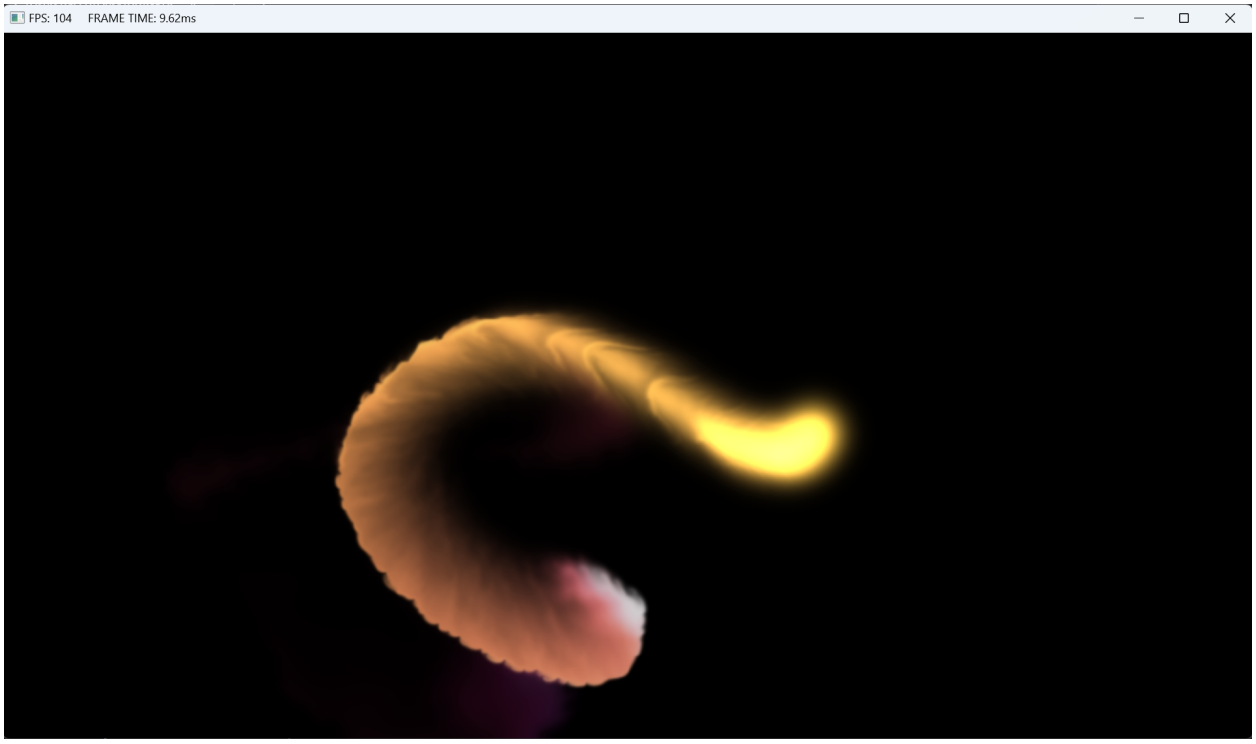
Un altra differenza rispetto alla simulazione interamente su CPU è che utilizziamo un solo array per ogni proprietà del fluido anziché due, questo perché possiamo sfruttare la sincronizzazione dei thread a nostro vantaggio. Per esempio uno schema generale dei kernel che compongono i vari step è il seguente.

```
__global__ computeValue(float* property)
{
      int x = blockIdx.x * blockDim.x + threadIdx.x;
      int y = blockIdx.y * blockDim.y + threadIdx.y;

      float cell = property[x,y];
      float leftCell = getLeftCell(x,y, property);
      float rightCell = getLeftCell(x,y, property);
      float bottomCell = getLeftCell(x,y, property);
      float topCell = getLeftCell(x,y, property);

      float newValue = computeNewValue(cell, leftCell, rightCell,
bottomCell, topCell)

      __syncthreads();
```

```
        property[x,y] = newValue;
}
```

Sebbene questa soluzione introduca del tempo di attesa dentro ai kernel, si è comunque rivelata più performante che utilizzare un secondo array per ogni valore.

Di seguito un esempio della simulazione che sfrutta anche la GPU.



Dettagli CUDA

Per l'implementazione di questo progetto sono state usate le seguenti tecniche messe a disposizione da CUDA.

Constant memory
Poiché l’applicazione è una simulazione fisica, vi sono molte costanti che fungono da coefficienti di varie formule, come la densità, la viscosità, la vorticità. Queste sono ideali da salvare in Constant Memory visto che verranno accedute spesso e in maniera read-only dai vari kernel

Shared memory
I kernel che si occupano di calcolare la diffusione e la pressione devono iterare diverse volte al fine di ottenere un risultato più fisicamente corretto. Possiamo quindi sfruttare la shared memory per fare cache dei dati che vengono acceduti ad ogni iterazione; in più abbiamo che celle vicine accedono a molti degli stessi dati; pertanto la shared memory fornisce un’efficienza ancora maggiore.

Stream
I dati della nostra simulazione non lasciano mai la memoria del device, in quanto come abbiamo visto, una volta che la simulazione completa un time step il risultato viene scritto nella texture direttamente dal device. Pertanto non abbiamo modo di sfruttare gli stream per sovrapporre trasferimento di memoria con computazione. Vi è però un punto della simulazione dove due kernel possono essere eseguiti in parallelo in quanto questi lavorano su due dati separati, nello step della diffusione.
Abbiamo infatti che la diffusione viene calcolata sia per la velocità del fluido che per il colorante in maniera separata; possiamo quindi sovrapporre le computazioni utilizzando due stream diversi.

Valori della simulazione e performance

Dimensione della griglia

La dimensione della griglia è il valore che ha più impatto sulla performance e sulla correttezza della simulazione. Una granularità molto fine della griglia significa che la simulazione è più precisa, ma pone anche dei grossi costi computazionali. In generale abbiamo che una buona dimensione per la griglia è:

$$\frac{ScreenSize}{8} < GridSize < \frac{ScreenSize}{2}$$

Con una griglia compresa tra questo range di valori abbiamo un ideale trade-off tra correttezza della simulazione e performance.

Per la simulazione su CPU abbiamo una dimensione della finestra di 1024x1024 ed una dimensione della griglia di 128x128.
Per la simulazione GPGPU abbiamo una dimensione della finestra di 1920x1080 ed una dimensione della griglia di 860x540.

Numero di iterazioni

Il numero di iterazioni per il calcolo della diffusione e della pressione è un altro parametro che impatta sia la precisione che la performance della simulazione.
Per la simulazione su CPU sia la diffusione che la pressione hanno un numero di iterazioni pari a 5.
Per la simulazione GPGPU la pressione ha un numero di iterazioni pari a 30 mentre la diffusione ha un numero di iterazioni pari a 20.

Performance

Vediamo ora le performance delle due simulazioni. I dati sono stati registrati su una macchina con processore **Intel Core i9-12900k** e con **GPU NVIDIA GeForce RTX 3090 (CC 8.6)**. Per la simulazione su GPU abbiamo che il numero di thread per ogni blocco è di **16x16**, mentre il numero di blocchi è di **(GRID_WIDTH/16)x(GRID_HEIGHT/16)**.

Simulazione CPU

La simulazione per CPU è notevolmente più lenta rispetto a quella GPGPU. Di seguito sono riportati i risultati della simulazione

```
Average compute time: 46.58ms
Max compute time: 56.66ms
Min compute time: 42.34ms

Average FPS: 21
Advect compute time: 10.05ms
Vorticity compute time: 4.91ms
Diffuse compute time: 19.16ms
Force compute time: 1.77ms
Pressure compute time: 8.73ms
Project compute time: 1.44ms
```

Simulazione GPGPU

La simulazione GPGPU è molto più performante della sua controparte su CPU come ci si aspettava, nonostante la maggior dimensione della griglia e il più alto numero di iterazioni.

```
Average compute time: 8.99ms
Max compute time: 11.78ms
Min compute time: 7.96ms

Average FPS: 108
Advect compute time: 0.94ms
Theoretical Occupancy: 83.33%
Achieved Occupancy: 80.06%

Vorticity compute time: 0.34ms
Theoretical Occupancy: 83.33%
Achieved Occupancy: 80.32%
```

```
Diffuse compute time: 6.56ms
Theoretical Occupancy: 100%
Achieved Occupancy: 93.36%

Force compute time: 0.22ms
Theoretical Occupancy:100%
Achieved Occupancy: 91.09%

Pressure compute time: 0.60ms
Theoretical Occupancy: 100%
Achieved Occupancy: 90.90%

Project compute time: 0.11ms
Theoretical Occupancy: 83.33%
Achieved Occupancy: 78.18%

Paint compute time: 0.06ms
Theoretical Occupancy: 100%
Achieved Occupancy: 87.62%

Bloom compute time: 0.08ms
Theoretical Occupancy: 100%
Achieved Occupancy: 87.96%
```